PCT WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 7 : G06F 17/50, 17/60, 9/44 — A1

(11) International Publication Number: **WO 00/13113**

(43) International Publication Date: 9 March 2000 (09.03.00)

(21) International Application Number: PCT/DK99/00456

(22) International Filing Date: 27 August 1999 (27.08.99)

(30) Priority Data:

PA 1998 01084	27 August 1998 (27.08.98)	DK
PA 1998 01095	31 August 1998 (31.08.98)	DK
PA 1999 00277	1 March 1999 (01.03.99)	DK

(71)(72) Applicants and Inventors: ANDERSEN, Henrik, Reif [DK/DK]; Klirevænget 55, DK–2880 Bagsværd (DK). HULGAARD, Henrik [DK/DK]; Koldinggade 20, 1. tv., DK–2300 Copenhagen Ø (DK). LICHTENBERG, Jacob [DK/DK]; Lindegårdsvej 12A 1. tv., DK–2920 Charlottenlund (DK). MØLLER, Jesper [DK/DK]; Frederikssundsvej 14A 4. mf., DK–2400 Copenhagen NV (DK).

(74) Agent: PLOUGMANN, VINGTOFT & PARTNERS A/S; Sankt Annæ Plads 11, P.O. Box 3007, DK–1021 Copenhagen K (DK).

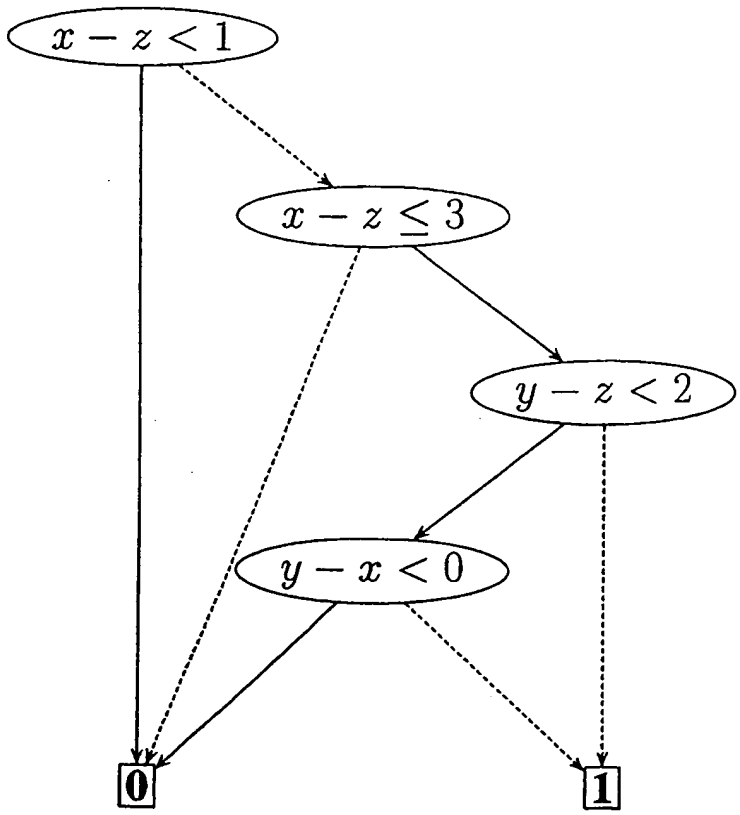
(81) Designated States: AE, AL, AM, AT, AT (Utility model), AU, AZ, BA, BB, BG, BR, BY, CA, CH, CN, CR, CU, CZ, CZ (Utility model), DE, DE (Utility model), DK, DK (Utility model), EE, EE (Utility model), ES, FI, FI (Utility model), GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MD, MG, MK, MN, MW, MX, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, SK (Utility model), SL, TJ, TM, TR, TT, UA, UG, US, UZ, VN, YU, ZA, ZW, ARIPO patent (GH, GM, KE, LS, MW, SD, SL, SZ, UG, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GW, ML, MR, NE, SN, TD, TG).

Published
With international search report.

(54) Title: A DATA STRUCTURE AND ITS USE

(57) Abstract

A data structure and its use in for example representation, analysis and verification of systems comprising continuous variables. Continuous variables arise in many areas of computer science and mathematics as for example timers or clocks in real–time controllers and digital circuits, sensors in embedded systems, counters in concurrent protocols, variables in confuguration problems, and scheduling times in planning and optimization problems. The data structure can represent and decide validity of first order propositional formulas over difference constraints or linear inqualities. The data structure can be used in symbolic model checking of concurrent timed systems modeled as timed automata, timed Petri nets or timed guarded commands. The data structure is preferably embodied as a decision diagram similar to binary decision diagrams (BDDs).



FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AL	Albania	ES	Spain	LS	Lesotho	SI	Slovenia
AM	Armenia	FI	Finland	LT	Lithuania	SK	Slovakia
AT	Austria	FR	France	LU	Luxembourg	SN	Senegal
AU	Australia	GA	Gabon	LV	Latvia	SZ	Swaziland
AZ	Azerbaijan	GB	United Kingdom	MC	Monaco	TD	Chad
BA	Bosnia and Herzegovina	GE	Georgia	MD	Republic of Moldova	TG	Togo
BB	Barbados	GH	Ghana	MG	Madagascar	TJ	Tajikistan
BE	Belgium	GN	Guinea	MK	The former Yugoslav Republic of Macedonia	TM	Turkmenistan
BF	Burkina Faso	GR	Greece			TR	Turkey
BG	Bulgaria	HU	Hungary	ML	Mali	TT	Trinidad and Tobago
BJ	Benin	IE	Ireland	MN	Mongolia	UA	Ukraine
BR	Brazil	IL	Israel	MR	Mauritania	UG	Uganda
BY	Belarus	IS	Iceland	MW	Malawi	US	United States of America
CA	Canada	IT	Italy	MX	Mexico	UZ	Uzbekistan
CF	Central African Republic	JP	Japan	NE	Niger	VN	Viet Nam
CG	Congo	KE	Kenya	NL	Netherlands	YU	Yugoslavia
CH	Switzerland	KG	Kyrgyzstan	NO	Norway	ZW	Zimbabwe
CI	Côte d'Ivoire	KP	Democratic People's Republic of Korea	NZ	New Zealand		
CM	Cameroon			PL	Poland		
CN	China	KR	Republic of Korea	PT	Portugal		
CU	Cuba	KZ	Kazakstan	RO	Romania		
CZ	Czech Republic	LC	Saint Lucia	RU	Russian Federation		
DE	Germany	LI	Liechtenstein	SD	Sudan		
DK	Denmark	LK	Sri Lanka	SE	Sweden		
EE	Estonia	LR	Liberia	SG	Singapore		

A data structure and its use

Field of the Invention

The present invention relates to a data structure and its use in for example representation, analysis and verification of systems containing continuous variables. The data strucure can be used for example to analyse real-time controllers, software, digital circuits, embedded systems, concurrent protocols, and solve configuration, optimisation and planning problems.

Background of the Invention

A difference decision diagram or DDD is a data structure for symbolically representing logical combinations of difference constraints (i.e., inequalities between a difference of two real-valued variables and a constant). The invention is embodied as a computer program implementing the data structure. The core of the invention is not the computer program in itself which is just one of many possible embodiments of the invention, but a description of the data structure and a number of effective algorithms and methods for manipulating the data structure.

The data structure was invented in a project developing methods for improving the quality of embedded systems. Embedded systems are computer programs embedded in larger systems and are often used to control the behavior of the system as for example in airbags, anti-blocking brakes or railroad safety systems. These methods are based on a technique called model checking which uses exhaustive testing of all possible states in the system to ensure that only certain good states are reachable. Until recently, exhaustive testing was considered infeasible because of the extremely large number of states. But new technological breakthroughs has changed this viewpoint. Real industrial problems with considerable complexity can be formally verified today. This has become possible by using compact data structures for representing large state spaces and efficient algorithms for manipulating them. The breakthrough for Boolean systems has enabled industrial circuit manufacturers such as Intel, Motorola and IBM to exploit these methods to improve the quality of their products and thus avoiding costly bugs in their designs (the division bug in one of Intel's Pentium processors, sold in more than a million copies, has been estimated to cost more than hundred million dollars).

Model checking has proved successful on systems with only Boolean variables, but it is still an open problem how to efficiently verify systems with (non-Boolean) discrete or continuous variables. In

such systems, integer or real valued variables play a crucial role in the correctness of the system. Examples are timing aspects of a digital circiut or temperature in an embedded system. It is contemplated that the data structure can extend the positive results from Boolean systems to systems with non-Boolean variables.

Systems with discrete and continuous variables become more and more prevalent and this has resulted in an increased demand for tools and methodologies to assist in the design, validation and test of such systems. Formal methodologies for reasoning about non-Boolean systems must contain a model including both the discrete and continuous behavior of the system. The need to represent both discrete and continuous values cause many verification algorithms to revert to use multiple data structures. This results in problems when relating, for example, control and data. As a consequence, state-of-the-art techniques for analyzing systems with time, modeled for example as timed automata, are only capable of analyzing systems with a handful of timers and a few thousand states.

The data structure can be applied not only in verification of safety properties of timed systems, but also in analysing a wide range of other problems. Difference constraints can model timing constraints, and by using logical connectives such as disjunction and existential quantification, difference constraints can also be used to represent and solve a number of planning problems such as optimal usage of a production plant, scheduling problems, economic planning problems and transport planning.

Difference constraints can also express interval constraints on variables which combined with the logical connectives is useful in configuration tools. Configuration tools are used to assist in the assembly of complex products or safety critical systems such as cars, railroad safety systems, vending machines, trains or PCs specialized to solve a specific task. The market for configuration tools is in strong growth and estimated to be at least 3 billion dollars per year. Experiments with industrial examples show that an approach based on the data structure results in a considerable improvement in performance. Thus, the data structure can extend the application areas of configuration tools to, for instance, online configuration on the Internet in connection with e-commerce.

Summary of the Invention

In a first aspect, the invention relates to an acyclic data structure comprising:

- a number of nodes comprising
  - at least a first and a second pointer pointing to other nodes,

- an expression comprising at least one inequality with at least one variable, the expression being adapted to result in one, of at least two disjoint outcomes, each pointer representing one of the outcomes, the number of pointers corresponding to the number of outcomes of the expression,

- at least one terminal node,
- at least one node pointing to the at least one terminal node,

the expressions being ordered according to predetermined criteria, the pointers of a first node comprising an expression of a first, lower order pointing to nodes comprising expressions of second orders, the second orders being higher than the first order.

In the present context, "acyclic" means that no incidents may occur so that, when following the pointers of the structure, will a path from a given node exist back thereto.

Also, in this context, the term "expression" will mean a mathematical expression comprising at least one variable, optionally one or more constants, possibly arithmetic operators, such as $+$, $-$, $*$, and $/$, and at least one comparison operator, such as $<$, $>$, $\leq$, or $\geq$.

One advantage of the ordering of the expressions may be seen when performing operations on one or more structures. When e.g. combining two structures, a search for nodes having a given representation is quick, as the ordering of the expressions will define where in the structure such a node could appear.

In the present context, "disjoint" preferably means that the expression, no matter the value(s) of the at least one variable, will always provide a unique outcome so that it is clear which pointer represents the outcome.

Naturally, the structure being acyclic, one or more roots are preferably present, the root(s) normally being a node not being pointed to. However, a root may be chosen within the structure, when that part of the structure pointed to by the root represents the interesting part of the structure. In fact, if the structure is optimally and totally reduced, the root may actually be a terminal node — such as in the situation where the structure — or the interesting part thereof — has been reduced to only that terminal node.

A terminal node is a node not pointing to any other nodes. In the present context, the terminal nodes may be adapted to represent constant expressions, such as the constants "false", "true", or "17".

Depending on the manner in which the structure has been constructed, a number of "local" incidents are preferably avoided in order to save space and in order to optimize the operation of the generation of the structure or the use thereof.

Thus, firstly, preferably, the data structure is at least substantially free from incidents of nodes where: the first and second pointers of a first node point to a second and a third node, respectively, the second pointer of the second node points to the third node, the expressions of the first and second nodes relate to the same variables, and the variable values fulfilling or not fulfilling the expression of the first node being comprised in the variable values fulfilling or not fulfilling the expression of the second node.

Secondly, the data structure is preferably at least substantially free from incidents of nodes where all pointers of a node point to the same node.

Thirdly, the data structure is preferably at least substantially free from incidents of nodes where two nodes exist having identical expressions and having pointers pointing to the same nodes, where the first pointers of the two nodes point to the same node, and where the second pointers of the two nodes point to the same node. Normally, this would be the situation pair-wise for all pointers of the nodes.

Preferably, the terminal nodes are adapted to represent Boolean values "true" and "false". This greatly simplifies the structure and the operation thereof. This provides a very versatile method and structure in that it provides the possible use of characteristic functions for representing sets and relations, and quite complicated operations, such as comparisons, unions, etc. may be performed without requiring enumeration of the elements of the sets.

Preferably, the expressions in the nodes except the terminal nodes all contain at least one inequality. The more uniform a structure, the easier to handle.

Preferably, the disjoint outcomes of the expressions constitute "true" or "false", and wherein each node comprises two pointers. The fact that all nodes (except for the terminal nodes) have the same type of information rendering the representation of the structure compact as well as simplifying the operations performed on the structure.

When the at least one inequality is a linear inequality, a large versatility is obtained within a large number of applications, such as economic models, optimization problems, and planning problems.

Most preferably, the inequalities are difference constraints, which are useful when analyzing concurrent software, embedded systems, real-time systems, hardware, and in timing analysis.

There exists a number of disadvantageous incidents in acyclic structures which make making enquiries to the structure more complicated.

Thus, the data structure is preferably at least substantially free from incidents of nodes where, when following a path from one node via one or more pointers to a second node, there exists no set of variable values fulfilling a combined expression obtained by, for each node entered, the expression therein having to provide the outcome corresponding to the pointer of the node pointing to the next node.

In this manner, superfluous nodes may be removed, and the presence of a given terminal node in the structure now guarantees that a set of variable values exists that provides the outcome leading to the terminal node.

Also, the data structure is preferably at least substantially free from incidents of pairs of paths, starting in the same starting node and ending in the same ending node, where, a single path may be generated starting in the starting node and ending in the ending node, the same variables values fulfill the combined expression obtained when following the single path from the starting node to the ending node as fulfill a disjunction of the pair of paths.

It is contemplated that, when the structure fulfills this criterion, the same structure will emerge independently on how the structure has been built from e.g. an analysis of a system — i.e. the structure will be a canonical representation of the system.

In a second aspect, the invention relates to a method of generating a data structure as described above and representing a system having a number of variables, the method comprising:

a) determining the variables,

b) defining a number of entities in the system, the entities defining relations between variables,

c) defining criteria for ordering the expressions,

d) representing each relation by:

- defining a number of different expressions each comprising at least one inequality with at least one variable, and each expression being adapted to result in one of at least two disjoint outcomes,
- associating each expression with a node, the node having:
  - at least a first and a second pointer adapted to point to other nodes,

- the number of pointers of the node corresponding to the number of outcomes of the expressions,

- ordering the expressions associated with the nodes in accordance with the defined criteria so that the pointers of a node comprising an expression of a lower order points to nodes comprising expressions of higher orders so as to generate an entity data structure representing the corresponding entity, and

e) combining the entity data structures to generate the data structure.

In the present context, a "variable" of the system is a part thereof which may vary — typically over time. This may be values of timers, delays of gates, or values of variables of software, values representing a physical phenomenon or measure, such as temperature, flux or the like.

When determining the number of variables, normally one would limit oneself to the variables that influence the part of the system, which is interesting.

Normally, the entities will be parts of the system through which variables may interact, such as gates in a circuit, statements in a program, or the like.

It has been found that the selected criteria for ordering of the expressions has significance on the final size of the structure as well as the simplicity and computational load of the operations performed thereon.

The combination of the individual entity data structures into the data structure is performed while retaining the order of the expressions. In fact, due to the ordering, the step of combining the entity structures is simplified, as will become clear below.

Having combined the entity structures and obtained the data structure, global relations between entities are obtained from the local relations between entities. Thus, one or more functional properties of the system such as safety properties, liveness properties, minimum or maximum values of variables, and satisfying variable assignments can be determined on the basis of the combined data structure.

In one situation, the combination of the entity data structures comprises a number of steps in each of which a number of entity data structures are combined, each step comprising:

a) in the system, determining a relationship between the entities represented by the entity data structures and a mathematical operation determined by the relationship,

b) generating a new data structure by: generating an operator node representing the mathematical

operation and having a number of pointers pointing to the entity data structures.

One advantage of this manner of combining the structure is that the operator nodes need not be converted into "normal" nodes, as they may be optimized out of the structure.

One manner of optimizing the structure is one wherein:

- a first node is identified, all pointers of which point to the same, second node,
- all pointers pointing to the first node are pointed to the second node, and
- the first node is removed.

Another manner is one wherein:

- two nodes are identified having identical expressions and having pointers pointing to the same nodes, where the first pointers of the two nodes point to the same node, and where the second pointers of the two nodes point to the same node,
- pointing all pointers pointing to a first of the two nodes to the other of the two nodes, and
- deleting the first node.

Preferably, a set of predetermined reduction rules are repeatedly applied to the operator nodes in order to remove operator nodes from the data structure so as to simplify the structure at a point in time before converting the operator nodes into "normal" nodes.

Most preferably, the pointers of the nodes would point pairwise to the same nodes so that the function of the two nodes is identical, and the first node may be omitted when all pointers pointing thereto are redirected to the second node. In order to finally convert the operator nodes to "normal" nodes, the method preferably further comprising the step of:

- identifying an operator node having pointers pointing to more than two data structures,
- replacing the identified operator node by a group of operator nodes, each operator node in the group having two pointers, the group of operator nodes pointing to the more than two data structures.

The preferred manner of converting an operator node into "normal" nodes is one comprising the steps of:

a) identifying an operator node having pointers pointing to two data structures comprising only terminal nodes or nodes the expressions of which represent inequalities,

b) replacing the identified operator node and the data structures pointed to thereby by a new data structure generated by performing the following procedure relating to the two data structures:

c)
- if the lowest order node of the first data structure and the lowest order node of the second data structure comprise identical expressions,
  - generating a new node having an expression identical thereto,
  - generating a first new data structure from the data structures pointed to by the first pointers of the two lowest order nodes by performing step c),
  - having the new node's first pointer point at the first new data structure,
  - generating a second new data structure from the data structures pointed to by the second pointers of the two lowest order nodes by performing step c),
  - having the new node's second pointer point at the second new data structure,
- if the lowest order node of the first data structure and the lowest order node of the second data structure comprise different expressions,
  - generating a new node having an expression identical to that of the two nodes having the lowest order,
  - generating a first new data structure from the data structures pointed to by the first pointer of the node having the lowest order and that node not having the lowest order by performing step c),
  - having the new node's first pointer point at the first new data structure,
  - generating a second new data structure from the data structures pointed to by the second pointer of the node having the lowest order and that node not having the lowest order by performing step c),
  - having the new node's second pointer point at the second new data structure,
- if the lowest order node of one of the data structures comprises an expression, and the other data structure is a terminal node,
  - generating a new node having an expression identical to that of the node comprising an expression,
  - generating a first new data structure from the data structures pointed to by the first pointer of the node comprising an expression and the terminal node by performing step c),

- having the new node's first pointer point at the new data structure,
- generating a second new data structure from the data structures pointed to by the second pointer of the node comprising an expression and the terminal by performing step c),
- having the new node's second pointer point at the second new data structure,

- if the two data structures are terminal nodes, performing the mathematical operation of the operator node between the terminal nodes and generating a data structure consisting of a terminal node representing the result of the operation.

Instead of introducing the operator nodes, the structures may be combined directly without the use of special-purpose nodes. One manner of combining two structures is one wherein the combination of the entity data structures comprises:

a) in the system determining a relationship between the two entities represented by the two data structures and a mathematical operation determined by the relationship,

b) generating a new data structure by:

- if the lowest order node of the first data structure and the lowest order node of the second data structure comprise identical expressions,
  - generating a new node having an expression identical thereto,
  - generating a first new data structure from the data structures pointed to by the first pointers of the two lowest order nodes by performing step b),
  - having the new node's first pointer point at the first new data structure,
  - generating a second new data structure from the data structures pointed to by the second pointers of the two lowest order nodes by performing step b),
  - having the new node's second pointer point at the second new data structure,
- if the lowest order node of the first data structure and the lowest order node of the second data structure comprise different expressions,
  - generating a new node having an expression identical to that of the two nodes having the lowest order,
  - generating a first new data structure from the data structures pointed to by the first pointer of the node having the lowest order and that node not having the lowest order by performing step b),
  - having the new node's first pointer point at the first new data structure,

- generating a second new data structure from the data structures pointed to by the second pointer of the node having the lowest order and that node not having the lowest order by performing step b),
- having the new node's second pointer point at the second new data structure,

- if the lowest order node of one of the data structures comprises an expression, and the other data structure is a terminal node,
  - generating a new node having an expression identical to that of the node comprising an expression,
  - generating a first new data structure from the data structures pointed to by the first pointer of the node comprising an expression, and the terminal node by performing step b),
  - having the new node's first pointer point at the new data structure,
  - generating a second new data structure from the data structures pointed to by the second pointer of the node comprising an expression and the terminal by performing step b),
  - having the new node's second pointer point at the second new data structure,
- if the two data structures are terminal nodes, performing the mathematical operation between the terminal nodes and generating a data structure consisting of a terminal node representing the result of the operation,

repeating steps a) and b) until only a single data structure remains.

In fact, this method of generating a structure or combining two data structures using a mathematical operation is a basic operation which may be used for other purposes than merely generation of data structures. For example the method is also useful for altering structures in order to prepare them for analysis/test. Thus, this more basic operation may comprise:

a) generating the new data structure by:

- if the lowest order node of the first data structure and the lowest order node of the second data structure comprise identical expressions,
  - generating a new node having an expression identical thereto,
  - generating a first new data structure from the data structures pointed to by the first pointers of the two lowest order nodes by performing step a),
  - having the new node's first pointer point at the first new data structure,

- generating a second new data structure from the data structures pointed to by the second pointers of the two lowest order nodes by performing step a),
- having the new node's second pointer point at the second new data structure,

- if the lowest order node of the first data structure and the lowest order node of the second data structure comprise different expressions,
  - generating a new node having an expression identical to that of the two nodes having the lowest order,
  - generating a first new data structure from the data structures pointed to by the first pointer of the node having the lowest order and that node not having the lowest order by performing step a),
  - having the new node's first pointer point at the first new data structure,
  - generating a second new data structure from the data structures pointed to by the second pointer of the node having the lowest order and that node not having the lowest order by performing step a),
  - having the new node's second pointer point at the second new data structure,
- if the lowest order node of one of the data structures comprises an expression, and the other data structure is a terminal node,
  - generating a new node having an expression identical to that of the node comprising an expression,
  - generating a first new data structure from the data structures pointed to by the first pointer of the node comprising an expression and the terminal node by performing step a),
  - having the new node's first pointer point at the new data structure,
  - generating a second new data structure from the data structures pointed to by the second pointer of the, node comprising an expression and the terminal by performing step a),
  - having the new node's second pointer point at the second new data structure,
- if the two data structures are terminal nodes, performing the mathematical operation between the terminal nodes and generating a data structure consisting of a terminal node representing the result of the operation.

In both the more specific case of generating data structures and in the more general of simply altering structures, it is preferred that the mathematical operations are chosen from the group consisting of

Boolean operators or combinators, such as AND, OR, NOT, and XOR, where the terminal nodes are given one of the values "true" and "false".

Especially in the more specific case, the mathematical operations are preferably binary operations, and the nodes comprising expressions are preferably generated with a first and a second pointer so as to be able to point at two other nodes, the second pointer being used, if the expression, given a set of variable values, is true, and the first pointer if the expression is false.

A number of methods exist for altering thus generated data structures in order to prepare the structures for certain analyses.

A first such method to existentially quantify out a variable is one comprising the steps of

a) identifying all paths leading from a root to a "true" terminal node,

b) for each path, constructing a difference bound matrix obtained from a combined expression obtained by, for each node entered in the path, the expression therein having to provide the outcome corresponding to the pointer of the node pointing to the next node,

c) solving the all pairs shortest path problem for each difference bound matrix,

d) removing in each matrix the row and column corresponding to a predetermined variable,

e) constructing a path from each matrix, and

f) combining all the paths by a disjunction using the above more general method using Boolean operators.

In this manner, a variable may be removed from the system in order to remove the constraints thereon. This may be interesting for use in e.g. an assignment.

Preferably a path is obtained from each matrix by a method where the construction step comprises, for each entry in the matrix, generating a node having a difference constraint corresponding to the variables of the row and column and the constant of the entry, and subsequently combining the resulting nodes by conjunction.

Especially advantageous is a method where the solving step comprises, for each matrix, solving the difference bound matrix by the algorithm of Floyd-Warshall performing only relaxation steps involving the predetermined variable.

Another, often more efficient, method for existentially quantifying out a variable comprises the steps of:

a) determining a variable,

b) generating a new data structure by:

- if the data structure is a terminal node then the result is said terminal node,
- if the lowest order node of the data structure does not comprise an expression containing the variable,
  - generating a first new data structure from the data structure pointed to by the first pointer of the node by performing step b),
  - generating a second new data structure from the data structure pointed to by the second pointer of the node by performing step b),
  - generating a new node having an expression identical to the expression of the node,
  - having the new node's first pointer point at the first new data structure,
  - having the new node's second pointer point at the second new data structure,
- if the lowest order node of the data structure comprises an expression containing the variable,
  - generating a first new data structure from the data structure pointed to by the first pointer of the node by performing a relaxing step with the negation of the node's expression as the constraining expression and then performing step b),
  - generating a second new data structure from the data structure pointed to by the second pointer of the node by performing a relaxing step with the node's expression as the constraining expression and then performing step b),
  - generating the resulting data structure as the disjunction of the first and the second new data structure.

A new data structure can then be obtained and used for further analysis.

The relaxing step with a given variable and a constraining expression which is either a lower or an upper bound on the variable, is preferably carried out by a method generating a new data structure by:

a)
- if the data structure is a terminal node then the result is said terminal node,
- if the lowest order node of the data structure comprises an expression containing the variable,

- generating a first new data structure from the data structure pointed to by the first pointer of the node by performing step a),
- generating a second new data structure from the data structure pointed to by the second pointer of the node by performing step a),
- if the constraining expression is an upper bound on the variable and the expression of the node is also an upper bound on the variable,
  * constructing a new expression without the variable obtained by combining conjunctively the constraining expression and the negation of the expression of the node,
  * generating the resulting data structure as the disjunction of
    · the negation of the expression of the node conjuncted with the first new data structure and the new, and
    · the expression of the node conjuncted with the second new data structure,
- if the constraining expression is an upper bound on the variable and the expression of the node is a lower bound on the variable,
  * constructing a new expression without the variable obtained by combining conjunctively the constraining expression and the expression of the node,
  * generating the resulting data structure as the disjunction of
    · the negation of the expression of the node conjuncted with the first new data structure, and
    · the expression of the node conjuncted with the second new data structure and the new expression,
- if the constraining expression is a lower bound on the variable and the expression of the node is an upper bound on the variable,
  * constructing a new expression without the variable obtained by combining conjunctively the constraining expression and the expression of the node,
  * generating the resulting data structure as the disjunction of
    · the negation of the expression of the node conjuncted with the first new data structure, and
    · the expression of the node conjuncted with the second new data structure and the new expression,
- if the constraining expression is a lower bound on the variable and the expression of the node is also a lower bound on the variable,

* constructing a new expression without the variable obtained by combining conjunctively the constraining expression and the negation of the expression of the node,
* generating the resulting data structure as the disjunction of
  - the negation of the expression of the node conjuncted with the first new data structure and the new expression, and
  - the expression of the node conjuncted with the second new data structure,

- if the lowest order node of the data structure does not comprise an expression containing the variable,
  - generating a first new data structure from the data structure pointed to by the first pointer of the node by performing step a),
  - generating a second new data structure from the data structure pointed to by the second pointer of the node by performing step a),
  - generating a new node having an expression identical to the expression of the node,
  - having the new node's first pointer point at the first new data structure,
  - having the new node's second pointer point at the second new data structure.

Using this method of relaxing a variable with respect to a given constraint a modified data structure can be obtained which can be used for further analysis.

Another method for removing a variable from the data structure comprises

- interchanging the terminal nodes "true" and "false",
- removing the variable using one of the methods above for performing existential quantification,
- interchanging the terminal nodes "true" and "false".

Using this method a variable can be universally quantified away from the data structure resulting in a data structure representing possible solutions to the remaining variables that are independent of the values of the chosen variable. This allows for the analysis of dependencies among variables in the data structure.

A method which is especially useful when using the data structure to represent relations and predicates which must be applied on different expressions is a method for replacing, in the data structure, a first variable x with the sum of a second, different variable y and a constant c comprising:

- constructing a second data structure by conjugating the initial data structure with a data structure comprising a conjunction of a first node comprising a difference constraint relating to $x - y \leq c$, and a second node comprising a difference constraint relating to $x - y \geq c$,
- combining the first and the second data structures by the Boolean operation of conjunction,
- removing x using one of the above methods.

When modeling dynamically changing systems it is important to be able to change the values of the variables within the data structure. This can be done advantageously by a method for replacing, in the data structure, a first variable x with the sum of a second, different variable y and a constant c comprising:

- removing x from the data structure,
- constructing a second data structure by conjugating the initial data structure with a data structure comprising a conjunction of a first node comprising a difference constraint relating to $x - y \leq c$, and a second node comprising a difference constraint relating to $x - y \geq c$,
- combining the first and the second data structures by the Boolean operation of conjunction.

In the special situation where the desired change to a variable is an increment or decrement of the value of that variable an efficient and advantageous method comprises:

- in each expression comprising a predetermined variable, replacing the variable by the same variable added with a predetermined constant.

In many applications it is essential to be able to determine the maximum and minimum bounds on all variables in a data structure. A method for achieving this comprises:

- identifying all paths leading from a root to a "true" terminal node,
- for each path, constructing a difference bound matrix obtained from a combined expression obtained by, for each node entered in the path, the expression therein having to provide the outcome corresponding to the pointer of the node pointing to the next node,
- solving the all pairs shortest path problem for each difference bound matrix,

- generating a maximum matrix from the difference bound matrices and having the same dimensions as the difference bound matrices by, for each entry in the maximum matrix, selecting the largest value in the difference bound matrices relating to the same entry, and
- obtaining information from the maximum matrix.

Using this method minimum and maximum delay times can be computed in for instance timed systems.

Often some of the nodes in a data structure are redundant. It can therefore be advantageous to reduce the data structure. When the nodes contain difference constraints, one such method for removing infeasible paths from a data structure comprises:

- for each path in the data structure from a root node to a terminal node:
  - for each node in the path, determining whether a set of variable values exists fulfilling a combined expression obtained by, for each node between the root node and the actual node, the expression therein having to provide the outcome corresponding to the pointer of the node pointing to the next node, and if no such set of variable values exists removing the pointer in the path pointing to the actual node.

Preferably, the determining step is performed according to the Bellman-Ford algorithm where, for each node in the path, information relating to the nodes already visited is stored and re-used in subsequent nodes.

When the nodes contain more general expressions a preferred method for removing infeasible paths from a data comprises:

- for each path in the data structure from a root node to a terminal node:
- for each node in the path, determining whether a set of variable values exists fulfilling a combined expression obtained by, for each node between the actual node and the root node, the expression therein having to provide the outcome corresponding to the pointer of the node pointing to the next node,
- removing the pointer in the path pointing to the actual node, wherein the determining step is performed using linear real programming, such as the simplex algorithm, or using integer linear programming.

In applications where it is particularly important to get a fully reduced data structure a method can be applied, comprising the steps of:

a) identifying all paths leading from a root to a "true" terminal node,

b) for each path, constructing a difference bound matrix obtained from a combined expression obtained by, for each node entered in the path, the expression therein having to provide the outcome corresponding to the pointer of the node pointing to the next node,

c) solving the all pairs shortest path problem for each difference bound matrix,

d) constructing a path from each matrix by expressing the bounds of each entry as difference constraints on the variables corresponding to the entry and forming the conjunction of the difference constraints, and

e) generating an amended data structure by combining all the paths by a disjunction, and

f) for each node in the amended data structure in each path from the root to a "true" terminal node:

g) determining an initial expression from a combination of the expressions of the nodes in the path between the root and the actual node,

h) determining a conjunctive combination between the initial expression and an expression obtained by a disjunction between the data structures pointed at by the two pointers of the node,

i) determining a conjunctive combination between:

- the initial expression and
- a disjunction between
  - a conjunction between the expression of the actual node and the data structure pointed at by the pointer representing a fulfillment of the expression of the node,
  - a conjunction between the negation of the expression of the actual node and the data structure pointed at by the pointer representing a non-fulfillment of the expression of the node,

j) if the variable values fulfilling the combination h) and the combination i) are identical, replacing the actual node by the disjunction between the data structures pointed at by the two pointers of the actual node.

Having constructed a data structure it can be analyzed in order to determine properties of the system modeled by the data structure. One such very useful method assesses whether there exists any set of values for the variables that when starting in a root of the structure, would result in a path ending in a predetermined terminal node, the method comprising:

- inspecting whether the data structure consists of one terminal node only,
- if so, a positive answer is returned, if the only terminal node is the predetermined terminal node, and a negative answer is returned, if the only terminal node is not the predetermined terminal node,
- if not, a positive answer is returned.

Using this method it can be determined for instance whether a data structure has no solutions or contains all assignments of values for the variables as a solution. By building the data structure to reflect comparison of two systems, or the implication between a system and a property, this method makes it possible to decide whether two systems are equivalent or a given system satisfies a given property. In analyzing any of the earlier mentioned application areas, this is an essential and highly useful method.

An example of an assignment of values to variables leading to a given terminal node can be obtained automatically on a data structure with no infeasible paths by a method comprising:

- starting in the root of the structure and repeating the step of:
  - if the first pointer of the node points to a terminal node different from the predetermined terminal node, selecting the node pointed to by the second pointer, otherwise selecting the node pointed to by the first pointer,
- if the predetermined terminal node is found:
  - constructing the path from the root to the terminal node and deriving a combined expression obtained by, for each node entered, the expression therein having to provide the outcome corresponding to the pointer of the node pointing to the next node, and
  - solving the combined expression and deriving a set of values of the variables in the solution.

Another useful method to analyze a data structure assesses whether a given set of values for the variables when starting in a root of the structure, would result in a path ending in a predetermined terminal node, the method comprising:

- starting in a predetermined root of the structure and repeating the step of: if the node is a terminal node, returning the contents of the terminal node, otherwise, evaluating the expression of the node according to the set of variable values and continuing with the node pointed at by the pointer corresponding to the outcome of the expression.

Using this method it can efficiently be decided whether an assignment of values to the variables results in any particular value in the data structure, thus to determine for instance whether the system modeled contains a known undesired, or desired, state.

Timed automata is a popular model of timed systems. These can be advantageously analyzed using the data structure of this document by generating a data structure for analyzing a system modeled by timed automaton having a number states and clocks, wherein:

step a) comprises:

- determining a first set of variables to be used for the encoding of the states,
- determining a second set of variables to be used for the clocks,

step b) comprises:

- identifying transitions between states, a transition comprising a starting state, an ending state, a requirement to be fulfilled in order to enable the transition to take place, an action (updating the variables, advancing time, etc.) to be performed when the transition takes place, and a requirement of the clocks to be fulfilled after the transition has taken place,

step d) comprises:

- for each transition, generating a data structure representing the requirement to be fulfilled in order for the transition to be enabled,

step e) comprises:

- constructing a data structure representing the set of reachable states by:
- constructing a data structure R representing a set of initial states of the automaton,
- repeatedly:

* selecting a transition,
* generating an amended data structure R' by conjugating the data structure representing the requirement of said selected transition with R,
* generating an amended data structure R'' by, in R', updating variables in accordance with the actions of the transition,
* assigning R as the disjunction of R and R'',

until R is unchanged for all transitions,

where after inquiries may be made as to the existence of predetermined states of the automaton using any of the above methods.

Even more complex systems can be modeled and analyzed by a concurrent system of timed automata. These systems can be analyzed by generating a data structure for analyzing a concurrent system modeled by a composition of a number of timed automata each having a number of states and clocks, wherein:

step a) comprises:

- determining a first set of variables to be used for the encoding of the individual states of the automata,
- determining a second set of variables to be used for the individual clocks of the automata,
- determining a third and fourth set of variables to be used for encoding the new values of the variables from the first and second set such that there is a one-to-one correspondence between the variables in the first and third set, respectively in the second and fourth set,

step b) comprises:

- identifying non-idling transitions between states, a non-idling transition comprising a starting state, an ending state, a requirement to be fulfilled in order to enable the transition to take place, an action to be performed when the transition takes place, and a requirement of the clocks to be fulfilled after the transition has taken place,
- identifying idling transitions from a state to itself, comprising a requirement to be fulfilled when none of the requirements of the non-idling transitions are fulfilled on that state, an empty action, and a requirement of the clocks to be fulfilled after the transition has taken place,

step d) comprises:

- for each transition, generating a data structure over the four set of variables, representing a relation expressing the requirement to be fulfilled in order for the transition to be enabled using the first two set of variables, expressing the action to be performed when the transition takes place using the third and fourth set of variables, and expressing the requirement of the clocks using the third and fourth set of variables,
- generating a data structure A representing the advance time predicate using variables from the second and fourth set of variables,
- constructing a data structure T representing the set of transitions by:
  * defining a data structure T as a terminal node representing "true",
  * for each automaton:
    · defining a data structure U as a terminal node representing "false",
    · for each transition of the automaton, assigning to U the disjunction of U and the selected transition,
    · assigning to T the conjunction of T and U,
  * assigning to T the disjunction of the advance time predicate A and T,

step e) comprises:

- constructing a data structure representing the set of reachable states by:
  * constructing a data structure R representing a set of initial states of the automata,
  * repeatedly:
    · generating a data structure R' by conjugating T and R,
    · generating a data structure R'' by quantifying out all variables from the first and second set of variables,
    · generating a data structure R''' by replacing all variables from the third and fourth set of variables with the corresponding variable for the first and second set,
    · assigning to R the disjunction of R and R''',

  until R is unchanged,

where after inquiries may be made as to the existence of predetermined states of the automata.

Timed Petri nets is another popular model of timed concurrent systems. These systems can be analyzed by a data structure using a method that from a timed Petri net, which has a number of transitions and states, each state having a clock and an associated time delay interval, constructs the data structure with a method comprising the steps of:

step a) comprises:

- determining a first set of variables to be used for the encoding of the states,
- determining a second set of variables to be used for the clocks,

step b) comprises:

- identifying transitions between states, a transition comprising a starting state, an ending state, and a requirement to be fulfilled in order to enable the transition to take place, the identified transitions possibly including a transition that advances time,

step d) comprises:

- for each transition, generating a data structure representing the requirement to be fulfilled in order for the transition to be enabled,

step e) comprises:

- constructing a data structure representing the set of reachable states by:
  * constructing a data structure R representing an initial state of the Petri net,
  * repeatedly:
    · selecting a transition,
    · generating an amended data structure R' by conjugating the data structure representing the requirements of selected transition with R,
    · generating an amended data structure R'' by, in R', updating variables in accordance with the actions of the transition,
    · assigning R as the disjunction of R and R'',

  until R is unchanged for all transitions,

where after inquiries may be made as to the existence of predetermined states of the Petri net using any of the methods above.

The data structure thus generated can be used for analysis by any of the above methods.

A third popular model for analyzing a system uses min/max/linear constraints, the model having a number of nodes, each either being a "max" node, a "min" node or a "linear" node, and a number of constraints each pointing from one node to another, each constraint representing a time interval. This model can be analyzed by building a data structure with a method comprising the steps of:

step a) comprises:

- determining a set variables, one for each node,

step b) comprises:

- identifying constraints between nodes, a constraint comprising a starting node, an ending node, and a time delay,

step d) comprises:

- for each node, generating a data structure by representing a relation between the actual node, the nodes from which constraints point to the actual node, time intervals of those constraints, and the type of the actual node (min, max, or linear),

step e) comprises:

- constructing a data structure by performing the conjunction of the data structures generated in step d).

This method is particularly useful if the terminal nodes are adapted to represent "true" or "false", and the inequalities in the nodes are difference constraints. Information on the model can then be obtained the data structure using any of the above methods.

In economic models, planning problems, and various optimization problems the solution is expressed as Boolean combinations of linear inequalities. Such models can be analyzed by constructing a data structure and analyze it using any of the above methods. The construction can be performed by a method comprising the steps of:

- determining the linear inequalities,
- defining a number of different expressions, each comprising a linear inequality, and
- combining the data structures using the method for computing Boolean operators.

Inquiries to the data structure can then be obtained by any of the above methods.

Embedded systems, fault-tolerant systems, safety-critical systems, and concurrent compositions of any such systems can be advantageously analyzed by making a model using timed automata, timed Petri nets, or min/max/linear constraints models and proceed with one of the above methods for constructing a data structure from the model.

Furthermore, a range of other important problems can be addressed with the data structure. One example is the interface timing between two components which can be verified using a method comprising:

- modeling the interface timing of the two components or systems using a min/max/linear constraint model,
- analyzing the model according to the any of the above methods.

Another example is in the analysis of economical systems, operations research systems, transport systems, or planning problems, with a method comprising:

- modeling the system or problem using Boolean combinations of linear inequalities,
- analyzing the model according to any of the above methods.

A third example is for the analysis of the timing behavior of a combinational circuit, with a method comprising:

- modeling the gates of the circuit using a min/max/linear constraint model,
- analyzing the model according to any of the above methods.

A fourth example is the analysis of the timing behavior of combinational parts of a sequential circuit, with a method comprising:

- modeling the gates of the parts of the circuit using a min/max/linear constraint model,
- analyzing the model according to any of the above methods.

A fifth example is the analysis of the timing behavior of an asynchronous circuit, the method comprising:

- modeling the gates of the circuit using a timed Petri net,
- analyzing the model according to any of the above methods for Petri nets.

A sixth example is for analyzing a sequential or concurrent computer program, the method comprising

- modeling statements, such as assignments or conditional guards, as expressions containing inequalities in a data structure as defined earlier,
- achieving a model of the full program by:
  - combining the models of the individual statements, using manipulation algorithms comprising Boolean operators, quantifiers and/or substitutions, according to any of the above methods,
  - constructing a data structure R representing an initial state of the program,
  - repeatedly:
    * selecting a statement,
    * generating an amended data structure R' by conjugating the data structure representing the requirements of selected statement with R,
    * generating an amended data structure R'' by, in R', updating variables in accordance with the actions of the statement,
    * assigning R as the disjunction of R and R'',

    until R is unchanged for all statements,
- analyzing the program by analyzing R using any of above methods.

The preferred embodiment of the invention is a program for a computer, the program performing any of the methods above and storing the data structure in its memory or on its disk.

In the following, a preferred embodiment of the generation of the data structure as well as a preferred embodiment of the use thereof is described in relation to the drawings, wherein:

Figure 1 shows Milner's scheduler—a small example of a protocol for starting and detecting termination of N tasks.

Figure 2 shows a DDD for the expression $\phi = 1 \leq x - z \leq 3 \wedge (y - z \geq 2 \vee y - x \geq 0)$.

Figure 3 shows an (x, y)-plot for the DDD in Figure 2 for $z = 0$.

Figure 4 shows the runtimes for Milner's scheduler.

Figure 5 shows an example of a timed automaton (used in Example 3).

Figure 6 shows a DDD for the expression $x_2 - x_1 < 0$.

Figure 7 shows a DDD for the expression $x_2 - x_1 \leq 0$.

Figure 8 shows a DDD for the expression $x_2 - x_1 = 0$.

Figure 9 shows a DDD for the expression $x_2 - x_1 \geq 0$.

Figure 10 shows a DDD for the expression $x_2 - x_1 > 0$.

Figure 11 shows a DDD for the expression $x_2 - x_1 \neq 0$.

Figure 12 shows a graph with a negative-weight cycle.

Figure 13 shows an (x, y)-plot of $\exists x.\phi$ for $z = 0$ (for the expression ϕ in Figure 2).

Figure 14 shows a DDD for $\exists x.\phi$ (for the expression ϕ in Figure 2).

In Annex A, preferred embodiments are given for a number of algorithms for generating, amending, and reducing the present data structure as well as for deriving information relating to the systems modelled thereby. These algorithms are:

Algorithm 1: MK. Create a node corresponding to the ITE expression $x - y \lesssim c \rightarrow h, l$.

Algorithm 2: MKNORM. Crete a node where the variables not necessarily are normalized.

Algorithm 3: MKDIFFCSTR. Create a difference constraint of the form $x - y \sim c$, where $\sim$ is one of $\{<, \leq, =, \neq, >, \geq\}$.

Algorithm 4: APPLY. Combine two DDDs with a Boolean operator.

Algorithm 5: NOT. Negate a DDD.

Algorithm 6: FEASIBLE. Determine whether a constraint system has a feasible solution using the Floyd-Warshall algorithm.

Algorithm 7: REDUCE. Path-reduce a DDD.

Algorithm 8: FEASIBLE'. Determine whether a constraint system has a feasible solution using an incremental version of Bellman-Ford's algorithm.

Algorithm 9: INSERTCSTR. Insert a constraint in list keeping it squeezed.

Algorithm 10: UNSATISFIABLE, TAUTOLOGY, SATISFIABLE, FALSIFIABLE, EQUIVALENT, CONSEQUENCE. Determine functional properties of a DDD based on REDUCE.

Algorithm 11: ALLINFEASIBLE. Determine whether all **0**- or **1**-path in a DDD are infeasible.

Algorithm 12: EXISTSFEASIBLE. Determine whether some **0**- or **1**-path in a DDD is feasible.

Algorithm 14: UNSATISFIABLE, TAUTOLOGY, SATISFIABLE, FALSIFIABLE. Determine functional properties of a DDD based on ALLINFEASIBLE and EXISTSFEASIBLE.

Algorithm 15: ANYSAT. Create a satisfying variable value assignment for a DDD if it is not unsatisfiable.

Algorithm 16: EXISTS. Existential quantification of a variable in a DDD.

Algorithm 17: RELAX. Relaxation of a DDD with a difference constraint.

Algorithm 18: FORALL. Universal quantification

Algorithm 19: ASSIGN. Assignment operator

Algorithm 20: INCREMENT. Increment operator

Algorithm 21: REPLACEMENT. Replacement operator

Algorithm 22: HULL. Convex hull of a DDD.

Algorithm 23: MERGE. Merge all disjunctive vertices in a DDD.

Analyzing Concurrent Systems (Overview)

The following first part of the description illustrates the basic features of the invention by means of simple terms and accompanying figures. The second part to follow will deal with the theoretical background of the invention in more detail. The detailed second part is directed towards an example of the invention described as Difference Decision Diagrams. It should nevertheless be emphasized that the invention can be utilized for any system which can be modeled using Boolean combinations of relational expressions.

To analyze a system, such as a digital circuit or embedded software, the system needs to be modeled mathematically. The mathematical model can then be rigorously scrutinized either by a human or, more practically, by a computer program. The invention described herein is used to efficiently analyze a system through a mathematical model and thereby obtain answers to questions such as whether the system can reach an erroneous state or whether the model may reach a preferred state.

Timed guarded commands is an example of a mathematical notation used to model concurrent systems. A timed guarded command program consists of a number of timed guarded commands which

have the form

$$g \rightarrow \vec{v} := \vec{d}$$

where g is a guard (a Boolean predicate) and $\vec{v} := \vec{d}$ is a multi-assignment of n constants in $\vec{d}$ to n variables in $\vec{x}$. The constants and variables may be Booleans, integers or reals. Timed guarded commands are a powerful notation for modeling concurrent systems which contain non-Boolean domains. Popular models such as timed Petri nets and timed automata can be represented in a straightforward manner by a timed guarded command program.

Referring now to Fig. 1, a small example is described in order to illustrate the key aspects of the invention. The example is a model of a concurrent system called Milner's scheduler. The scheduler consists of N cyclers, connected in a ring, cooperating on starting and detecting termination of N tasks (these tasks are not further described). The scheduler must make sure that the N tasks are always started in order but they are allowed to terminate in any order. This is one of the properties that has to be shown to hold for the model. The cyclers attempt at fulfilling this by passing a token: the holder of the token is the only process allowed to start its task.

All cyclers are similar except that one of them has the token in the initial state of the system. We associate three Boolean variables c_i, h_i, and t_i with each cycler and use a global clock H to ensure that a cycler passes the token on to the following cycler within a bounded amount of time given by the interval $[H^l, H^u]$. This clock is modeled using a real-valued variable in the timed guarded command program. The variable c_i is used to denote whether the token is available for task i, variable h_i denotes whether cycler i has the token, and t_i denotes whether the task is running. The i^{th} cycler is described by two timed guarded commands and the task is modeled by a third guarded command:

$$\begin{array}{lll} c_i \wedge \neg t_i & \rightarrow H, t_i, c_i, h_i & := 0, \textbf{true}, \textbf{false}, \textbf{true} \\ h_i \wedge H^l \leq H \leq H^u & \rightarrow c_{(i \bmod N)+1}, h_i & := \textbf{true}, \textbf{false} \\ t_i & \rightarrow t_i & := \textbf{false}. \end{array}$$

The first timed guarded command expresses that if the token is available for the i^{th} cycler ($c_i = \textbf{true}$) and the i^{th} task is not running ($t_i = \textbf{false}$), then the token clock is reset ($H := 0$), the i^{th} task is started ($t_i := \textbf{true}$) and the cycler grabs the token ($c_i := \textbf{false}$ and $h_i := \textbf{true}$). The second command expresses that if the cycler has the token ($h_i = \textbf{true}$) and the clock H is within the interval $[H^l, H^u]$, then the token is passed on to the next cycler in the ring. The third guarded command expresses that the task may terminate at any point if it is running.

To complete the model of the scheduler, a timed guarded command for advancing time is needed. Such a command increments the clock H with some arbitrary positive amount δ:

$$\textbf{true} \rightarrow H := H + \delta\,.$$

Notice that since the right-hand side of the assignment is not a constant value and δ furthermore represents an arbitrary value, the above guarded command cannot be written down explicitly as a timing guarded command. Below, it is shown in detail how the advance time commands are represented.

To answer questions such as whether it always is the case that at most one cycler has the token, one needs to analyze the set of reachable states of the system. A single state is a pair (s, v) where s is a discrete state and v is the associated timing information (a clock assignment). For example, a state of a scheduler with $N = 2$ could be that $c_0 = \textbf{true}$, and the five other Boolean variables (h_0, t_0, c_1, h_1, and t_1) all are false and the associated timing information is that clock H has the value 3.1415135. This state is thus represented by (s, v) where $s = \langle \textbf{true}, \textbf{false}, \textbf{false}, \textbf{false}, \textbf{false}, \textbf{false} \rangle$ and $v = \langle 3.1415927 \rangle$. To check a given property, one can determine the set of all states reachable from the initial state of the system. This set is denoted R. Observe that R is not a finite set since the clock H is modeled using a real-valued variable: for example R may contain infinitely many states (s, v) where H is between 0.1 and 0.2.

To analyze timed systems, clock valuations are grouped into sets. This allows the state space of a timed system to be represented as a finite set of pairs (s, V) of discrete states and their associated set of clock valuations. For example, a set of clock valuations can be represented as $V = 0.1 \leq H \leq 0.2$. It turns out that when constructing the set of reachable states, all groups of clock valuations can be expressed using the following grammar:

$$\psi ::= x - y \leq d \mid \neg\psi \mid \psi_1 \wedge \psi_2 \mid \psi_1 \vee \psi_2\,, \tag{1}$$

where x and y are real-valued clock variables and d is a constant.

The states of a timed system can be further grouped by combining the discrete state s with the group of clock valuations V. This is done by expanding the above grammar to also include Boolean variables:

$$\psi ::= x - y \leq d \mid \neg\psi \mid \psi_1 \wedge \psi_2 \mid \psi_1 \vee \psi_2 \mid b\,,$$

where b is a Boolean variable. Notice that the grammar does not contain inequalities of the form $x \leq d$. To express such constraints, a new variable z is introduced. This variables denotes "zero"

or "current time" and is used to express all constraints of the form $x \leq d$ as $x - z \leq d$. Using the z-variables, the set of clock valuations $V = 0.1 \leq H \leq 0.2$ is expressed in the above grammar as $z - H \leq -0.1 \wedge H - z \leq 0.2$.

The discrete state is combined with the timing information by expressing the discrete state using the Boolean variables and combine the information using conjunction:

$$(c_0 \wedge \neg h_0 \wedge \neg t_0 \wedge \neg c_1 \wedge \neg h_1 \wedge \neg t_1) \wedge (z - H \leq -0.1 \wedge H - z \leq 0.2).$$

Assume two states (s, V) and (s', V') are both represented using formulas ϕ and ϕ' as just described. The formula $\phi \vee \phi'$ is then a formula that represents the set of states $\{(s, V), (s', V')\}$. This way of representing sets of states using formulas makes it possible to construct the set of reachable states by manipulations of formulas. The set of reachable states R is computed using the following algorithm (a standard fixed-point iteration):

$Q \leftarrow \phi_0$
$R \leftarrow Q$
while SATISFIABLE(Q) **do**
  $Q' \leftarrow$ NEXT(Q)
  $Q \leftarrow Q' \wedge \neg R$
  $R \leftarrow R \vee Q$

The initial state of the system is represented by the formula ϕ_0. The formula Q represents the "frontier" of the states, i.e., the set of newly discovered states. The formula R represents the set of reachable states of the system. The procedure SATISFIABLE determines whether a given formula is satisfiable, i.e., whether there exists values for the variables which makes the formula true. The procedure NEXT(ϕ) determines a formula representing the set of states reachable by executing any timed guarded command or advancing time from a state satisfying ϕ. This procedure is described in detail in the following.

The efficiency of the above algorithm is determined by how efficiently one can represent the formulas Q and R and how efficiently one can implement the procedures SATISFIABLE(Q) and NEXT(Q). The invention described herein provides a compact data structure for representing formulas of the above form and provides efficient algorithms for implementing the procedures SATISFIABLE(Q) and NEXT(Q). Thus, in an embodiment the invention enables a highly efficient analysis of timed systems.

The data structure "difference decision diagrams" (DDDs) is an example of an embodiment of the

invention. A DDD is a directed acyclic graph with two terminals **0** and **1** and a set of non-terminal nodes. Each non-terminal node in a DDD comprises a test expression (a difference constraint) and has two outgoing pointers called the high- and low-branch which are drawn with solid and dashed lines, respectively. The high-branch is followed when the test expression evaluates to true; the low-branch when the test expression evaluates to false.

As an example of a DDD consider the following expression ψ over x, y and z:

$$\psi = 1 \leq x - z \leq 3 \ \wedge \ (y - z \geq 2 \ \vee \ y - x \geq 0) .$$

Figure 2 shows a DDD for the formula ψ, and Fig. 3 shows the values of x and y for which ψ is true.

The invention provides methods for performing the operations needed in the procedures SATISFIABLE(Q) and NEXT(Q). One method describes how to construct the DDD for the formula $\phi_1 \oplus \phi_2$ where $\oplus$ is an arbitrary binary Boolean operator and given DDDs for the formulas ϕ_1 and ϕ_2. Another method describes how to construct the DDD for the formula $\exists x.\phi$ given the variable x and a DDD representing the formula ϕ. Finally, the invention provides methods for determining whether a given DDD represents a satisfiable formula. These methods are all described in detail in the following.

Returning to the scheduler example, the set of reachable states has been constructed for an increasing increasing number of cyclers, N, using the above algorithm. The results are shown in the Fig. 4. The first column shows the number of cyclers, and the following three columns show the CPU time (in seconds) to build the reachable state space using the current tools KRONOS and UPPAAL. The last column shows the CPU time for constructing the set of reachable states when using an embodiment of the invention (for example DDDs). The results were obtained on a Pentium II PC with 64 MB of memory. A '–' denotes that the analysis did not complete within an hour. Clearly, the invention enables a dramatic improvement in the size of systems that can be analyzed compared with current state-of-the-art tools.

After constructing the set of reachable states R, it is straightforward to determine properties of the system. For example, to determine whether a state exists in which both cycler i and j ($i \neq j$) hold the token, the DDD for the formula $R \wedge h_i \wedge h_j$ is construct. If and only if this formula is satisfiable does there exists reachable states in which both cyclers have the token. Similarly, to test whether a property P holds in all states, the DDD for the formula $R \wedge \neg P$ is constructed. If and only if this formula is satisfiable does there exists a state in which P does not hold. More general properties can also be determined with the methods described by this invention — this is further explained in the following.

Analyzing Timed Systems (Detailed Description)

In the following a detailed description of how to analyze timed system modeled as guarded commands is given.

A *timed guarded command program* G comprises a tuple (B, C, T, I), where B is a set of Boolean variables, C is a set of continuous variables called *clocks*, T is a set of *timed guarded commands*, and I is a *state invariant*. A timed guarded command $t \in T$ has the form $g \rightarrow \vec{v} := \vec{d}$, where g is a guard and $\vec{v} := \vec{d}$ is a multi-assignment of n constant values $\vec{d} \in (\mathbb{B} \cup \mathbb{R})^n$ to Boolean variables and clocks $\vec{v} \in (B \cup C)^n$. Guards and state invariants are expressions ϕ constructed from the following grammar:

$$\phi ::= \mathbf{false} \mid \mathbf{true} \mid x \sim d \mid x - y \sim d \mid b \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \exists b.\phi \mid \exists x.\phi\,, \tag{2}$$

where $x, y \in C$ are clocks, $b \in B$ is a Boolean variable, $d \in \mathbb{R}$ is a constant, and $\sim$ is a relational operator from $\{\leq, <, =, \neq, >, \geq\}$. The symbols $\neg$ (negation), $\wedge$ (conjunction) and $\exists$ (existential quantification) have their usual meaning.

Example 1 An example of a program is $G = (\{b\}, \{x, y\}, T, I)$, where T contains the two guarded commands

$$b \wedge (1 \leq x \leq 3) \rightarrow b \quad := \mathbf{false}$$
$$b \wedge (7 \leq x \leq 9) \rightarrow b, y := \mathbf{false}, 0$$

and the state invariant is $I = \big(b \Rightarrow (x \leq 9)\big) \wedge \big(\neg b \Rightarrow (x \neq 5)\big)$.

Transitional Semantics of Timed Guarded Commands

A *state* of the program $G = (B, C, T, I)$ is an interpretation (i.e., a value assignment) of the Boolean variables and the clocks. For each Boolean variable $b \in B$, $s(b) \in \mathbb{B}$ denotes the interpretation of b in the state s, and for each clock $x \in C$, $s(x) \in \mathbb{R}$ denotes the interpretation of x in the state s. The notation $s[x := y + d]$ is used to denote the state s' equivalent to s except that $s'(x) = s(y) + d$. A state (and sets of states) can be represented by an expression ϕ of the form (2). The state s *satisfies* an expression ϕ, written $s \models \phi$, if ϕ evaluates to true in the state s, and $[\![\phi]\!]$ denotes the set of states that satisfy ϕ.

The semantics of a timed guarded command program $G = (B, C, T, I)$ is a *transition system* $(S, \rightarrow)$,

where S is the set of states of the program, and $\rightarrow$ is the transition relation. In each state, the program can either execute a command $t \in T$ if its guard is true (a discrete transition) or let time pass δ time units (a timed transition). Executing a command changes the value of some or all of the variables (according to the multi-assignment), and letting time pass uniformly increases the values of all clocks by δ. The notation $s \xrightarrow{t} s'$ is used for a discrete transition from the state s to s' obtained by executing the command t, and the notation $s \xrightarrow{\delta} s'$ for a timed transition obtained by increasing all clocks by δ. The discrete transition $\xrightarrow{t}$ for a timed command $t \in T$ of form $g \rightarrow \vec{v} := \vec{d}$ is defined by the following rule:

$$\frac{s \models g \quad s[\vec{v} := \vec{d}] \models I}{s \xrightarrow{t} s[\vec{v} := \vec{d}]}. \tag{3}$$

The timed transition $\xrightarrow{\delta}$ for advancing all clocks by δ is defined by the following rule:

$$\frac{\delta \geq 0 \quad \forall \delta'.0 \leq \delta' \leq \delta : s[\vec{c} := \vec{c} + \delta'] \models I}{s \xrightarrow{\delta} s[\vec{c} := \vec{c} + \delta]}, \tag{4}$$

where $\delta, \delta' \in \mathbb{R}$, $\vec{c}$ denotes a vector of all clocks in C, and $\vec{c} + \delta$ denotes the vector where δ is added to the clocks in $\vec{c}$.

Example 2 Consider the timed guarded command program G from Example 1 and let s be a state satisfying $\neg b \wedge (x < 5)$. There are infinitely many timed transitions from s in the transition system for G, but none of these timed transitions leads to a state where $x \geq 5$ because the state invariant $\neg b \Rightarrow (x \neq 5)$ must hold continuously.

Encoding Timed Automata

Timed guarded command programs can be used to model popular notations for timed systems such as timed automata. A timed automaton over a set of clocks consists of a set of locations, a set of events, and a set of timed transitions. Each location is associated with a location invariant over the clocks, and each timed transition from location l to location l' is labeled with an event a and has a guard g over the clocks. Furthermore, each of the timed transitions has a set of clocks $\{\vec{c}\}$ to be reset when the timed transition is fired:

$$l \xrightarrow[\{\vec{c}\}]{a,\, g} l'.$$

A timed automaton can be encoded as a timed guarded command program. Each location is encoded

as a Boolean variable. In a shared variable model as ours, the presence of an event from an alphabet Σ can be modeled by a global *event variable* e taking on any of the values in Σ. This variable can for instance be encoded using a logarithmic number of Boolean variables. Each timed transition in the automaton corresponds to a timed guarded command:

$$l \wedge e_a \wedge g \rightarrow l, l', \vec{c} := \text{false}, \texttt{true}, \vec{0}.$$

The guard of the command is the guard of the timed transition g conjoined with the source location l of the timed transition and a condition e_a requiring the event variable e to have the value $a \in \Sigma$. The multi-assignment assigns false to the source location l and true to the destination location l' of the timed transition and resets the relevant clocks.

Example 3 Figure 5 shows an automaton over the clocks $\{x, y\}$ with two locations and two timed transitions. Encoding this automaton as a timed guarded command program yields the program G from Example 1 when ignoring the event a and encoding the two locations l_1 and l_2 logarithmically using a Boolean variable b.

Analyzing Timed Guarded Commands

To verify properties of a timed guarded command program $G = (B, C, T, I)$, the corresponding transition system $(S, \rightarrow)$ is analyzed symbolically. That is, given a set of states represented by a formula ψ, one determines a formula that represents the set of states reachable by executing timed guarded commands according to the inference rule (3) or by advancing time according to the inference rule (4). In the following it is shown that this formula is obtained by manipulations entirely within the logic (1).

Any expression ϕ generated by the grammar (2) can be represented by a *difference constraint expression* ϕ_z of the form (1). The expression ϕ_z is obtained by introducing a new variable z (denoting "zero") and performing the following three steps: First, encode each Boolean variable $b_i \in B$ in ϕ as a difference constraint $x_i - x_i' < 0$, where $x_i, x_i' \in C$ are clocks only used in the encoding of b_i. Second, replace each occurrence of a constraint of the form $x \sim d$ in ϕ with the difference constraint $x - z \sim d$. Third, express each difference constraint of the form $x - y \sim d$ in terms of the relational operator $\leq$.

Two useful operators on difference constraint expressions are defined: replacement and assignment. *Replacement* syntactically substitutes all occurrences of a variable x by a variable y plus a constant

d in an expression ψ, denoted by $\psi[y + d/x]$. If x and y are different variables, the replacement $\psi[y + d/x]$ can be expressed in the grammar (1) as $\exists x.(\psi \wedge (x - y = d))$. Otherwise, $\psi[x + d/x]$ is defined as $\psi[t/x][x + d/t]$, where t is a variable different from x and not occurring in ψ. *Assignment* gives a variable x the value of a variable y plus a constant d, denoted by $\psi[x := y + d]$. If x and y are different variables, the assignment $\psi[x := y + d]$ is expressed in the grammar (1) as $(\exists x.\psi) \wedge (x - y = d)$. Otherwise, the assignment $\psi[x := x + d]$ is defined as $\psi[x - d/x]$ (which might seem counter-intuitive). Assignment and replacement of Boolean variables are defined in the standard way.

To formally expresses the symbolic manipulations, a useful shorthand is introduced: $[\![\psi]\!]_z$ is used as a shorthand for $[\![\exists z.(\psi \wedge z = 0)]\!]$; that is, $[\![\psi]\!]_z$ is the set of states that satisfy ψ when z is equal to 0. It is easy to prove that $[\![\phi]\!] = [\![\phi_z]\!]_z$, for any ϕ. Eliminating the constraints of the form $x \sim d$ from the grammar in (2) makes it possible to add δ to all clocks simultaneously by decreasing the common reference-point z by δ:

$$[\![\phi[\vec{c} := \vec{c} + \delta]]\!] = [\![\phi_z[z := z - \delta]]\!]_z \,. \tag{5}$$

Furthermore, as will be shown in the following, the set of states reachable by advancing time by any value δ can be computed by an existential quantification of z.

Reachability Analysis

Given an expression ψ of the form (1) representing a set of states $[\![\psi]\!]_z \subseteq S$, an expression representing the set of states reachable from $[\![\psi]\!]_z$ can be determined. The set of states reachable by firing the timed guarded command t from any state in $[\![\psi]\!]_z$ is determined by the function $\text{NEXT}_{\text{discrete}}(\psi, t)$. The function restricts ψ to the subset where the guard g holds, performs the assignment of the constants $\vec{d}$ to the variables $\vec{v}$, and restricts the resulting set to the subset where the state invariant I holds:

$$\text{NEXT}_{\text{discrete}}(\psi, g \to \vec{v} := \vec{d}) = (\psi \wedge g_z)[\vec{v} := \vec{d}] \wedge I_z \,, \tag{6}$$

where the assignment $[\vec{v} := \vec{d}]$ is a shorthand for $c_i := z + d_i$ for each of the clocks c_i in $\vec{v}$ and $b_i := d_i$ for each of Boolean variables b_i in $\vec{v}$. The set of states that can be reached from the set $[\![\psi]\!]_z$ by firing any timed guarded command in T is given by:

$$\text{NEXT}_{\text{discrete}}(\psi) = \bigvee_{t \in T} \text{NEXT}_{\text{discrete}}(\psi, t) \,. \tag{7}$$

The z-variable plays a central role when determining the set of states that can be reached from $[\![\psi]\!]_z$ by firing a timed transition. Time is advanced by changing the reference-point from z to z' with $z' \leq z$ since decreasing the reference-point by δ corresponds to increasing the values of all clocks by δ. Often the system will restrict the valid choices for z' by requiring that the state invariant holds in z' and at all intermediate points in time. This is expressed by the predicate

$$P_{\text{next}} = (z' \leq z) \wedge I_{z'} \wedge \forall z''.\big((z' < z'' \leq z) \Rightarrow I_{z''}\big)\,.$$

If the state invariant I_z only expresses upper bounds on the clocks, the universal quantification is implied by $I_{z'}$ and can be omitted.

Now, to advance time by δ in all states $[\![\psi]\!]_z$, the reference-point z is decreased by δ: $\psi[z := z - \delta]$ which can also be written as $\big(\exists z.(\psi \wedge z' = z - \delta)\big)[z/z']$. The set of states reachable from $[\![\psi]\!]_z$ that also satisfy P_{next} is given by $\exists z.\big(\psi \wedge (z - z' = \delta) \wedge P_{\text{next}}\big)[z/z']$. Thus, the set of states reachable from $[\![\psi]\!]_z$ by advancing time by an *arbitrary* amount is given by

$$\textsc{Next}_{\text{timed}}(\psi) = \bigvee_{\delta \in \mathbb{R}} \exists z.\big(\psi \wedge (z - z' = \delta) \wedge P_{\text{next}}\big)[z/z'] = \exists z.(\psi \wedge P_{\text{next}})[z/z']\,. \qquad (8)$$

That is, time is advanced in a set of states by performing a single existential quantification.

Example 4 If the state invariant is $x \neq 5$, the predicate P_{next} is given by:

$$\begin{aligned} P_{\text{next}} &= (z' \leq z) \wedge (x - z' \neq 5) \wedge \forall z''.\big((z' < z'' \leq z) \Rightarrow (x - z'' \neq 5)\big) \\ &= (z' \leq z) \wedge \big((x - z' < 5) \vee (x - z > 5)\big)\,. \end{aligned}$$

Consider the set of states satisfying $\phi = (1 \leq x \leq 3) \vee (7 \leq x \leq 9)$. The set of states obtained by advancing time from ϕ is thus given by $[\![\textsc{Next}_{\text{timed}}(\phi_z)]\!]_z$, where:

$$\textsc{Next}_{\text{timed}}(\phi_z) = \exists z.(\phi_z \wedge P_{\text{next}})[z/z'] = (1 \leq x - z < 5) \vee (7 \leq x - z)\,.$$

A timed guarded command $t \in T$ is called *urgent* if it is required to fire instantaneously whenever the guard becomes true. Modifying P_{next} to handle urgent commands is straightforward: Given a set $T' \subseteq T$ of urgent timed guarded commands, we let U denote the predicate:

$$U = \bigvee_{g \to \vec{v} := \vec{d} \in T'} g\,.$$

Consider a state $s \in [\![\psi]\!]_z$. A timed transition $s \xrightarrow{\delta} s'$ can only fire if there are no urgent transitions enabled in s. Thus, an additional requirement is added to P_{next} ensuring that no urgent transitions are enabled when advancing time (except in the endpoint), i.e., the revised P_{next} becomes:

$$P_{\text{next}} = (z' \leq z) \wedge I_{z'} \wedge \forall z''.\big((z' < z'' \leq z) \Rightarrow (I_{z''} \wedge \neg U_{z''})\big)\,.$$

If the urgency predicate does not refer to z, P_{next} is simplified to

$$P_{\text{next}} = (z' \leq z) \wedge I_{z'} \wedge \forall z''.\big((z' < z'' \leq z) \Rightarrow I_{z''}\big) \wedge \neg U_z\,.$$

The functions defined in (7) and (8) form the basis for constructing the set of reachable states symbolically. Let $\textsc{Next}(\psi)$ be a function which determines the set of states which can be reached by firing either a discrete or a timed transition from a state in $[\![\psi]\!]_z$:

$$\textsc{Next}(\psi) = \textsc{Next}_{\text{discrete}}(\psi) \vee \textsc{Next}_{\text{timed}}(\psi)\,.$$

The set of states reachable from $[\![\psi]\!]_z$, denoted $\textsc{Reachable}(\psi)$, is the least fixed point of the function $F(X) = \psi \vee \textsc{Next}(X)$, which can be determined using a standard fixed-point iteration. Detecting that a fixed point has been reached is done by checking that two successive approximations ψ_i and ψ_{i+1} are semantically equivalent (i.e., that $\psi_i \Leftrightarrow \psi_{i+1}$ is a tautology). It is well known that there exists (contrived) timed systems where the computation of the fixed point does not terminate, for example if the difference between two clocks increase ad infinitum. As in the traditional analysis of timed automata, it is possible to determine subclasses of timed guarded commands for which termination is ensured.

Example 5 Consider again the program from Example 1. The set of states reachable from $\phi = b \wedge (x = y = 0)$ is $[\![\textsc{Reachable}(\phi_z)]\!]_z$, where:

$$\begin{aligned}\textsc{Reachable}(\phi_z) \;=\; & \big(b \wedge x = y \wedge x - z \leq 9\big) \\ \vee\; & \big(\neg b \wedge ((x = y \wedge 1 \leq x - z < 5) \vee (7 \leq x - y \leq 9 \wedge 7 \leq x - z))\big)\,.\end{aligned}$$

Symbolic Model Checking

To perform symbolic model checking, for example of a TCTL formula, the set of states that can reach a given set $[\![\psi]\!]_z$ needs to be determined. The set of states that can reach $[\![\psi]\!]_z$ by firing any timed

guarded command $g \rightarrow \vec{v} := \vec{d}$ in T is given by

$$\text{PREV}_{\text{discrete}}(\psi) = \bigvee_{g \rightarrow \vec{v} := \vec{d} \in T} \left(\exists \vec{v}.(\psi \wedge \vec{v} = \vec{d})\right) \wedge g_z \wedge I_z \,, \tag{9}$$

where the expression $\vec{v} = \vec{d}$ is a shorthand for $c_i - z = d_i$ for each of the clocks c_i in $\vec{v}$ and $b_i \Leftrightarrow d_i$ for each of Boolean variables b_i in $\vec{v}$. The set of states that can reach $[\![\psi]\!]_z$ by advancing time is determined analogously to the forward case:

$$\text{PREV}_{\text{timed}}(\psi) = \exists z.(\psi \wedge P_{\text{prev}})[z/z'] \,,$$

where

$$P_{\text{prev}} = (z \leq z') \wedge I_{z'} \wedge \forall z''.\left((z < z'' \leq z') \Rightarrow I_{z''}\right) .$$

The set of states that can reach a state in $[\![\psi]\!]_z$ by firing either a discrete or a timed transitions is:

$$\text{PREV}(\psi) = \text{PREV}_{\text{discrete}}(\psi) \vee \text{PREV}_{\text{timed}}(\psi) \,.$$

Thus, the set of states that can reach a state satisfying ψ is constructed as the least fixed-point of the function $B(X) = \psi \vee \text{PREV}(X)$. Moreover, PREV can be used to perform symbolic model checking of TCTL. TCTL is a timed version of CTL obtained by extending the logic with an auxiliary set of clocks called *specification clocks*. These clocks do not appear in the model and are used to express timing bounds on the temporal operators. The atomic predicates of TCTL are difference constraints over the clocks from the model and the specification clocks. Semantically, the specification clocks become part of the state, they proceed synchronously with the other clocks but are not changed by the model. A specification clock u can be bound and reset by a *reset quantifier* $u.\psi$.

Symbolically, the set of states satisfying a given TCTL formula ψ can be found by a backward computation using a fixed-point iteration for the temporal operators. For instance, the set of states satisfying the formula $\psi_1 E U \psi_2$ is computed symbolically as the least fixed point of the function $B(X) = \psi_2 \vee (\psi_1 \wedge \text{PREV}(X))$. The set of states satisfying $u.\psi$ is computed symbolically as $\exists u.(\psi \wedge u - z = 0)$, i.e., the reset quantifier corresponds to restricting the value of u to zero and then remove it by existential quantification. The atomic predicates and the Boolean connectives correspond precisely to the corresponding difference constraint expressions.

Above the set of states has been determined using a constrained image approach. To compose systems

synchronously, as used for instance in timed automata, a timed guarded command program can be encoded using a *transition relation* R over "present-state" variables $V = B \cup C \cup \{z\}$ and the "next-state" variables $V' = \{v' : v \in V\}$ (as traditionally done in symbolic model checking of discrete systems but including the reference points z and z'). The relation R is constructed by combining the transitions of each automaton using disjunctions and then combining the automata using conjunctions. Thus, the parallel composition of a set of timed automata can be analyzed fully symbolically, i.e., both symbolically with respect to the parallel composition *and* with respect to the representation of sets of clock valuations and discrete states. Using a transition relation, well-known and very useful tricks from the work on BDDs, such as early variable quantification and partitioned representation of the transition relation are immediately applicable.

Difference Decision Diagrams

The preferred embodiment of the invention is a data structure called difference decision diagrams or DDDs where the nodes comprise difference constraints such as inequalities of the form $x - y \leq c$ or $x - y < c$, where x and y are integer or real-valued variables and c is a constant.

A *difference decision diagram* (DDD) is a directed acyclic graph. The node set comprises two terminals **0** and **1** with out-degree zero, and a set of non-terminal nodes with out-degree two. Each non-terminal node v comprises a difference constraint expression of the form $x - y < c \rightarrow h, l$ or $x - y \leq c \rightarrow h, l$, with the following attributes: $pos(v) = x$, $neg(v) = y$, $op(v) \in \{\text{LE}, \text{LEQ}\}$ (LE denoting the operator $<$, and LEQ denoting the operator $\leq$), $const(v) = c$, $high(v) = h$, and $low(v) = l$. The symbol $\lesssim$ is used to denote either $<$ or $\leq$. The edge set comprises the edges $(v, low(v))$ and $(v, high(v))$, where $v \in V$ is a non-terminal node.

A *root* in a DDD is a node that represents an expression of particular interest. Any node in a DDD can be a root.

A difference decision diagram represents a formula implicitly: Each non-terminal node corresponds to an if-then-else operator. The if-then-else operator $\alpha \rightarrow \psi_1, \psi_0$ is defined as $(\alpha \wedge \psi_1) \vee (\neg\alpha \wedge \psi_0)$,

where α is a Boolean expression. The meaning of a node (or a root) is defined recursively by:

$$\begin{aligned}
[\![0]\!] &\stackrel{\text{def}}{=} \text{false},\\
[\![1]\!] &\stackrel{\text{def}}{=} \text{true},\\
[\![v]\!] &\stackrel{\text{def}}{=} \begin{cases} x - y < c \rightarrow [\![h]\!], [\![l]\!] & \text{if } op(v) = \text{LE},\\ x - y \leq c \rightarrow [\![h]\!], [\![l]\!] & \text{if } op(v) = \text{LEQ}. \end{cases}
\end{aligned}$$

where $x = pos(v)$, $y = neg(v)$, $c = const(v)$, $h = high(v)$, and $l = low(v)$. In the following, two notational shorthands are frequently used: $var(v) = (pos(v), neg(v))$ and $bound(v) = (op(v), const(v))$. Adding two bounds (o_1, c_1) and (o_2, c_2) gives $(o_1 + o_2, c_1 + c_2)$, where $o_1 + o_2$ is LEQ if both o_1 and o_2 are LEQ and LE otherwise. Negating a bound (o, c) gives $(\neg o, -c)$, where $\neg$LE is LEQ and $\neg$LEQ is LE.

Figures 6–11 show some examples of DDDs, namely the six basic difference constraints $x_2 - x_1 \sim 0$, where $\sim$ is one of $\{<, \leq, =, \neq, >, \geq\}$. To make the figures easier to comprehend and appear more pleasant to the eye, a minus sign is shown between the variables, and in the figures $<$ is written instead of LE, and $\leq$ for LEQ. High-branches are drawn with solid lines, and low-branches are drawn with dashed lines.

Ordering

The expressions of the nodes in a DDD are ordered. Such an order can for example be constructed from an ordering of the variables $x_1, \ldots, x_n$ as follows. Assume that the variables are named so that they are ordered according to their indices:

$$x_1 \prec x_2 \prec \cdots \prec x_n.$$

Pairs of variables (x_i, x_j) of a node in a DDD are conveniently assumed to be *normalized*; that is, $x_i \succ x_j$. This does not restrict what can be represented with DDDs, because

$$x_j - x_i < c \rightarrow h, l \;=\; \neg(x_j - x_i < c) \rightarrow l, h, \;=\; x_i - x_j \leq -c \rightarrow l, h$$

and similarly for $x_j - x_i \leq c \rightarrow h, l$. With n variables there can be at most $n(n-1)/2$ normalized pairs of variables (x_i, x_j), which for instance can be ordered such that

$$(x_i, x_j) \prec (y_i, y_j) \quad \text{if and only if} \quad x_j \prec y_j \;\vee\; (x_j = y_j \;\wedge\; x_i \prec y_i),$$

that is

$$
\begin{aligned}
&(x_2, x_1) \prec (x_3, x_1) \prec (x_4, x_1) \prec \cdots \prec (x_n, x_1) \prec \\
&(x_3, x_2) \prec (x_4, x_2) \prec \cdots \prec (x_n, x_2) \prec \\
&\vdots \\
&(x_{n-1}, x_{n-2}) \prec (x_n, x_{n-2}) \prec \\
&(x_n, x_{n-1}).
\end{aligned}
$$

With, for example, four variables the ordering could be:

$$(x_2, x_1) \prec (x_3, x_1) \prec (x_4, x_1) \prec (x_3, x_2) \prec (x_4, x_2) \prec (x_4, x_3).$$

The two operators are preferably ordered such that $\texttt{LE} \prec \texttt{LEQ}$, and the constants are preferably ordered as usually in $\mathbb{Z}$ or $\mathbb{R}$. The 4-tuples $\big(pos(v), neg(v), op(v), const(v)\big)$ of attributes are then for example ordered lexicographically. An example is:

$$(x_2, x_1, \texttt{LE}, 0) \prec (x_2, x_1, \texttt{LEQ}, 0) \prec (x_5, x_1, \texttt{LEQ}, 1) \prec (x_5, x_2, \texttt{LE}, 0) \prec (x_4, x_3, \texttt{LE}, 0).$$

It is convenient to let the two terminals be greater than all non-terminals. In practice, it is convenient to define **0** and **1** to have all the attributes of the non-terminals, and let the variables of **0** and **1** be x_{n+1}.

Implementation

A DDD can be implemented as a data structure in a computer program. The nodes and pointers are stored in a global table in the computer's memory. Associated with each node is a set of attributes consisting of a mathematical expression comprising at least one inequality with at least one variable, and a number of pointers corresponding to the number of outcomes of the expression. If, for example, the expression is a difference constraint, the attributes of a node comprise at least two variables, an inequality operator, a constant, and two pointers.

Nodes and edges in a DDD can for example be stored as a graph G. Initially, G comprises the two terminal nodes **0** and **1**. A non-terminal node comprises at least six attributes of type

$$\textbf{Attr} \;=\; \textbf{Var} \times \textbf{Var} \times \{\texttt{LE}, \texttt{LEQ}\} \times \mathbb{D} \times V \times V.$$

Attributes of a node and the node itself are distinguished. The node is merely a unique identifier: an

index in a table of attributes. The edges of G are not stored explicitly, but implicitly via the attributes in the nodes. The following operations on the graph are used (*Graph* denotes the type of the graph G): $insert : Graph \times \mathbf{Attr} \to V$, $member : Graph \times \mathbf{Attr} \to \mathbb{B}$, and $lookup : Graph \times \mathbf{Attr} \to V$. The function $insert(G, a)$ creates a new node v in G with attributes a and returns v. The high- and low-branches must be nodes already in G, and the variables must be different. The function $member(G, a)$ returns true if G contains a node with attributes a. The function $lookup(G, a)$ returns the node in G that has attributes a. If G has no node with attributes a, the function is unspecified.

Insertion can be done in constant time. In practice, however, memory management must be taken into account: Memory must be allocated for the attributes and garbage collection is performed when memory becomes exhausted. These memory management functions can be implemented using standard techniques known to a person skilled in the art such that the expected cost for an insertion will be $O(1)$. The two other operations (*member* and *lookup*) can be done in expected constant time, using a hash table that maps attributes to nodes.

The following operations are easily implemented on the attributes of non-terminal nodes: $pos : V \to \mathbf{Var}$, $neg : V \to \mathbf{Var}$, $var : V \to \mathbf{Pair}$, $op : V \to \{\text{LE}, \text{LEQ}\}$, $const : V \to \mathbb{D}$, $bound : V \to \mathbf{Bound}$, $high : V \to V$, $low : V \to V$, where $\mathbf{Pair} = \mathbf{Var} \times \mathbf{Var}$, and $\mathbf{Bound} = \{\text{LE}, \text{LEQ}\} \times \mathbb{D}$. The type $\mathbf{Cstr} = \mathbf{Pair} \times \mathbf{Bound}$ is used to represent constraints. The implementation use n variables which are ordered. Each variable x_i is uniquely identified by the index i (i.e., $x_i = x_j$ implies $i = j$), so that the variable indices can be used to index matrices and arrays.

The algorithms MK and MKDIFFCSTR create DDDs for basic expressions. MK creates a DDD for an ITE expression $x - y \lesssim c \to h, l$, and MKDIFFCSTR creates a DDD for a difference constraint $x - y \sim c$, where $\sim$ is one of $\{<, \leq, =, \neq, >, \geq\}$.

ITE Expressions

The basic operation on the DDD data structure is MK, which creates a node for an ITE expression. The function MK only creates locally reduced nodes. Using MK as the only means for creating nodes in the DDD, will make sure that they are locally reduced. The function is preferably implemented as the computer program presented in Algorithm 1.

A pair of variables (x, y) is said to be *normalized* if $x \succ y$, i.e., x is after y in the variable ordering. In MK the pair of variables (x, y) must be normalized, and the node to be created must be ordered with respect to the high- and low-branches. The function MK consists of four steps: (1) if the domain

is $\mathbb{Z}$, weak upper bounds are used; (2) if G already contains an identical node, it is returned; (3) if the high- and low-branches are identical, one of them is used; and (4) if the test is obviously redundant, the low-branch is returned. Clearly, MK creates only nodes that are locally reduced, provided that the input is ordered.

If a pair of variables is not normalized, the function MKNORM shown in Algorithm 2 can for instance be used to normalize a constraint. If $x = y$, we return **0** if the upper bound is negative, and **1** otherwise (clearly, $x - x < 0$ is false, but all weaker bounds makes it true). Otherwise, we normalize the pair of variables, and negate the bound.

Difference Constraints

The function MKDIFFCSTR(x, y, o, c) shown in Algorithm 3 uses MK to create DDDs for the six types of difference constraints, see Fig. 6–11. The construction is dependent on the operator o, which is one of $\{$EQ, NEQ, LEQ, GEQ, LE, GR$\}$. In each case, the difference constraint can be expressed using LE or LEQ, or a combination of both. Furthermore, for $o =$ EQ and $o =$ NEQ, the two nodes are combined in the correct order, so that the resulting DDD is still ordered. In both cases the node with LE must be the topmost node, because LE $<$ LEQ.

Boolean Combinations

To combine DDDs with Boolean connectives the function APPLY shown in Algorithm 4 is preferably used. APPLY is based on five equivalences and uses the well-known technique of dynamic programming to avoid exponential running time.

Difference constraint expressions can be combined with conjunction, disjunction, implication and bi-implication, and can be negated. The APPLY algorithm allows any Boolean combination of two expressions to be performed. Conn denotes the set of all dyadic Boolean connectives. A function

$$eval : \mathrm{Conn} \times \{0,1\} \times \{0,1\} \rightarrow \{0,1\}$$

returns the result of combining two terminal nodes with the given Boolean connective. For example, $eval(\mathrm{AND}, 1, 1) = 1$.

APPLY is a generalization of the version used for reduced order binary decision diagrams, which is

based on the fact that any binary Boolean operator *op* distributes over the if-then-else operator:

$$(\alpha \to h, l)\ op\ (\alpha' \to h', l') \;=\; \alpha \to \big(h\ op\ (\alpha' \to h', l')\big), \big(l\ op\ (\alpha' \to h', l')\big)\,.$$

This equivalence provides a method to combine two DDDs with a Boolean connective. Reading the equivalence from left to right, it is seen that the Boolean connective can be moved down one level in the DDD. If this is continued, until both arguments of *op* are **0** or **1**, the Boolean expression can be evaluated and the appropriate result returned. In the above equation, α is the topmost constraint on the right-hand side, but also α' could be used:

$$(\alpha \to h, l)\ op\ (\alpha' \to h', l') \;=\; \alpha' \to \big((\alpha \to h, l)\ op\ h'\big), \big((\alpha \to h, l)\ op\ l'\big)\,.$$

If the two pairs of variables are equal, one or two of the branches can be evaluated directly. There are three cases depending on whether $\alpha \prec \alpha'$, $\alpha = \alpha'$, or $\alpha \succ \alpha'$:

$$(\alpha \to h, l)\ op\ (\alpha' \to h', l') = \begin{cases} \alpha \to (h\ op\ h'), (l\ op\ (\alpha' \to h', l')) & \text{if } \alpha \prec \alpha', \\ \alpha \to (h\ op\ h'), (l\ op\ l') & \text{if } \alpha = \alpha', \\ \alpha' \to (h\ op\ h'), ((\alpha \to h, l)\ op\ l') & \text{if } \alpha \succ \alpha'. \end{cases}$$

To avoid exponential running time, dynamic programming is preferably used to memorize the previously computed results. As it is well-known, a global hash-table H of type *HashTable* can be implemented having the following operations:

$$\begin{aligned} \mathit{insert}: &\quad \mathit{HashTable} \times (\mathbf{Conn} \times V \times V) \times V \to \mathbf{unit} \\ \mathit{member}: &\quad \mathit{HashTable} \times (\mathbf{Conn} \times V \times V) \to \mathbb{B} \\ \mathit{lookup}: &\quad \mathit{HashTable} \times (\mathbf{Conn} \times V \times V) \to V \end{aligned}$$

The function $\mathit{insert}(H, (op, u, v), r)$ creates a new entry $((op, u, v), r)$ in H where r is the result of computing APPLY(op, u, v). The function $\mathit{member}(H, (op, u, v))$ returns true if APPLY(op, u, v) has been computed previously. The function $\mathit{lookup}(H, (op, u, v))$ returns the result r of computing APPLY(op, u, v). If the result is not in the hash table, the function is unspecified.

The efficiency of APPLY can be further improved in the special cases, where one of the operands is true or false, or where the two operands are identical. The improved algorithm depends on the operator. For conjunction, for example, the following program fragment can be inserted before the first if-statement in APPLY:

```
if op = AND then
  if u = 0 ∨ v = 0 then
    return 0
  elsif u = 1 then
    return v
  elsif v = 1 ∨ u = v then
    return u
```

Note that the check $u = v$ is only syntactical. If u and v are semantically equivalent, but not syntactically the same, the expression will not be simplified. This is a consequence of the lack of canonicity of locally reduced DDDs. However, experiments with examples show, that APPLY runs approximately twice as fast with these optimizations.

Negation

A DDD u can be negated by using Algorithm 5, where NP1 is the binary operator that negates its first argument and discards the second.

Path Reducing

To further improve the representation of the DDD, the data structure can be *path-reduced.* In a path reduced DDD, all **0**- and **1**-paths are feasible. A path defines a constraint system S as the conjunction of all the constraints occurring when following the high- and low-branches in a path. As it is well-known, such a constraint system can be represented as a directed weighted graph and a solution found by solving a shortest path problem. Determining feasibility (i.e., the existence of a solution) of S corresponds to the non-existence of a negative-weight cycle in the constraint graph G_S induced by S. If the constraint graph is represented as a square matrix, the well-known FLOYD-WARSHALL algorithm can be used to find a negative-weight cycle with the algorithm FEASIBLE shown in Algorithm 6. The algorithm calls FLOYD-WARSHALL, and if any diagonal-element is negative, it returns false; otherwise, it returns true.

An efficient algorithm for performing path reduction is obtained using an *incremental* version of Bellman-Ford's single-source shortest paths algorithm. Before presenting the algorithm, it is shown how the Bellman-Ford algorithm can be used to determine whether a graph has a negative-weight cycle.

The Bellman-Ford algorithm uses a technique called *relaxation.* It makes $n - 1$ passes over the edges

in weighted graph G_S with n nodes, and in each pass all edges are relaxed once. A relaxation consists of updating a *distance array* d, where each entry d_i is an estimate on the minimal distance from a virtual node x_0 to the node x_i. Initially all estimates are set to (≤ 0). Each pass monotonically decreases the estimates in d, and if an estimate has not converged after $n-1$ passes, the graph has a negative-weight cycle. That is, if we in the n^{th} pass can relax *any* edge, the graph has a negative-weight cycle. Introducing the virtual node corresponds to adding a new variable x_0 to the constraint system and letting $x_i - x_0 \leq 0$ for all $i = 1, 2, \ldots, n$. Clearly, this does not change the feasibility of the system. If the system is feasible, each entry d_i is the minimal distance from x_0 to x_i.

The used incremental version of the Bellman-Ford algorithm has been modified at two points: (1) the number of passes to make is minimized, and (2) a better initial estimate than (≤ 0) is used. Firstly, the number of passes is minimized by stopping after the first pass that does not change any estimates in the distance array. Clearly, if a pass does not change the distance array, then the subsequent passes will not change it either, because estimates decrease monotonically. Secondly, the improved initial estimates are obtained as follows: After having run the Bellman-Ford algorithm on a graph, and having found that it has no negative-weight cycles, the distance array d contains the minimal distances from x_0 to all other nodes. Suppose, an extra edge $e = (x_i, x_j)$ is added to the graph. This edge may or may not change some or all of the estimates in d, but it will not cause any of the estimates to increase. Thus, the estimates can be reused. In most situations, adding an edge to a graph changes only some of the minimal distances, so there is a good chance that only a few extra passes is needed to recalculate d. For example, if the graph already has a path from x_i to x_j with weight less than or equal to the weight of e (i.e., e is redundant), then d contains the correct minimal distances, and the algorithm stops after one pass.

The incremental version of Bellman-Ford is used to test for feasibility in the algorithm REDUCE that path-reduces a DDD. The algorithm uses a list L in which each element is a pair $((x_i, x_j), (\lesssim c))$ denoting an edge from x_i to x_j with weight $(\lesssim c)$. The length of the list is limited to $n(n-1)$ by keeping it squeezed (i.e., removing consecutive tests on the same pair of variables). The set W contains the variables in the path from u to v, which gives a bound on the number of passes that have to be made. Very often, the number of variables in a path is much less than n.

Before discussing the incremental Bellman-Ford algorithm used in REDUCE, consider the graph in Fig. 12. Clearly, it has a negative-weight cycle (i.e., a cycle with weight less than (≤ 0)), but using the operation defined for upper bounds, this negative-weight cycle will not be detected neither with the incremental nor the original Bellman-Ford algorithm. To see why, recall that $(< 0) + (< 0) = (< 0)$. Thus, after one pass all estimates will be (< 0), and the relaxing of the edges is stopped and true is

returned (i.e., the graph has no negative-weight cycle). Informally speaking, the problem is that the number of $<$'s in the estimates is not counted. $(< 0) + (< 0)$ should give $(\ll 0)$, and $(\ll 0) + (< 0)$ should give $(\lll 0)$, etc.

To cope with this problem, a variable to count the number of $<$'s in a path is used. The bound $\leq$ counts as 0, and $<$ counts as 1. An estimate now becomes a pair of integers (δ_i, c_i) where δ_i denotes the number of $<$'s, and c_i is the weight of the path (without any $<$'s or $\leq$'s). This explains why all elements in d are initialized to $(0, 0)$ in the top level call. Adding an upper bound to an estimate and comparing two estimates are defined by:

$$\begin{aligned}
(\text{LEQ}, c_1) + (\delta, c_2) &\stackrel{\text{def}}{=} (\delta, c_1 + c_2) \\
(\text{LE}, c_1) + (\delta, c_2) &\stackrel{\text{def}}{=} (\delta + 1, c_1 + c_2) \\
(\delta_1, c_1) \prec (\delta_2, c_2) &\Leftrightarrow c_1 < c_2 \vee (c_1 = c_2 \wedge \delta_1 > \delta_2)
\end{aligned}$$

Algorithm 8 shows the incremental version of Bellman-Ford. At most $m = |W|$ passes are made, and if a pass does not change the distance array (i.e., *unchanged* is **true**), the algorithm stops. Notice that if the first relaxation in the loop leaves d unchanged, the first constraint in L is redundant. A relaxation consists of iterating over all elements in L and updating d if an edge makes an estimate better. An edge (x_i, x_j) with weight b makes an estimate (δ_j, c_j) better if $(\delta_j, c_j) \succ b + (\delta_i, c_i)$. Inserting constraints in a list maintaining it squeezed is done by the function INSERTCSTR shown in Algorithm 9.

Functional Properties

It is easy to write algorithms that check for tautology, satisfiability, etc., using REDUCE, see Algorithm 10. However, a preferred implementation is often based on algorithms that search for counter-examples. If u has a feasible 0-path, u is not a tautology; that is, the search can stop after encountering the first feasible 0-path. This observation is used in the algorithm ALLINFEASIBLE shown in Algorithm 11 which returns true if and only if all t-paths ($t \in \{0, 1\}$) in v are infeasible, stopping when it finds the first feasible t-path. Similarly, we define EXISTSFEASIBLE in Algorithm 12 which returns true if and only if there exists a feasible t-path in v. Algorithm 11 and 12 can then be used to test for functional properties as shown in Algorithm 14.

On some examples it is more efficient and therefore preferable to change the first three lines in Algorithm 11 to:

if $v = t$ **then**

return $\neg$FEASIBLE$'(|W|, Array(0,0), L)$
elsif $v \in \{0,1\}$ **then**
return true

And similarly for Algorithm 12.

Anysat

If a DDD is not unsatisfiable, a satisfying value assignment can be found for it. Recall that the distance array d for a feasible constraint system contains the minimal distances from x_0 to all other nodes. In terms of the constraint system, d_i corresponds to the least upper bound on $x_i - x_0$. Letting (arbitrarily) $x_0 = 0$, a feasible solution can be read off the constraint system:

$$x_i = \begin{cases} c & \text{if } d_i = (\delta, c) \wedge \delta = 0, \\ c - \epsilon & \text{if } d_i = (\delta, c) \wedge \delta > 0. \end{cases}$$

where ϵ is some sufficiently small, positive constant. The value ϵ is preferably chosen to be less than the minimal difference between any two different constants c_k and c_l in d.

ANYSAT shown in Algorithm 15 is almost identical to EXISTSFEASIBLE: If the path is feasible in the terminal case $v = \mathbf{1}$, an assignment a is constructed as described in the equation above and returned. If the path is not feasible, or if $v = \mathbf{0}$, the value $\perp$ is returned indicating that the path has no satisfying value assignment. In the non-terminal case, ANYSAT is called on the high-branch and—if that does not result in a satisfying assignment—ANYSAT is called on the low-branch.

Quantifiers

It is generally important to be able to perform quantification of variables in order to make enquiries to the values represented by a DDD. Existential quantification of a variable x in an expression ϕ removes x from ϕ. Before presenting the algorithm EXISTS to perform this job, a small example is shown to illustrate what the algorithm should do. Figure 3 illustrates the set of solutions represented by a DDD. Quantifying out x in the expression $\phi = 1 \leq x - z \leq 3 \ \wedge \ (y - z \geq 2 \ \vee \ y - x \geq 0)$ yields $\exists x.\phi = y - z \geq 1$, see Fig. 13. Here, the constraint $y - z \geq 1$ does not occur explicitly in ϕ, but implicitly because of $y - x \geq 0$ and $x - z \geq 1$, see Fig. 14.

The existential quantification of a variable x in a DDD u consists of removing all nodes comprising

x from u, but keeping all the implicit constraints induced by x among the other variables.

To compute $\exists x.(x_i - x_j \lesssim c \rightarrow h, l)$, two cases must be considered: If x is different from both x_i and x_j, the quantifier can be pushed down one level in the DDD:

$$\exists x.(x_i - x_j \lesssim c \rightarrow h, l) \;=\; x_i - x_j \lesssim c \rightarrow \exists x.h, \exists x.l \qquad \text{if } x \notin \{x_i, x_j\}.$$

If x is equal to x_i or x_j, all paths in h and l with $x_i - x_j \leq c$ and $x_i - x_j > c$, respectively, are *relaxed* and the results are combined with disjunction:

$$\begin{aligned}\exists x.(x_i - x_j \leq c \rightarrow h, l) \;=\;& \exists x.\textsc{Relax}(h, x, x_i - x_j \leq c) \\ & \vee \exists x.\textsc{Relax}(l, x, x_j - x_i < -c) \qquad \text{if } x \in \{x_i, x_j\}.\end{aligned}$$

The case for $\exists x.(x_i - x_j < c \rightarrow h, l)$ is analogous. The algorithm for existential quantification is shown in Algorithm 16.

If x is equal to x_i, relaxation of a path p with a constraint $x_i - x_j \lesssim c$ consists of adding a new constraint $x_i' - x_j \lesssim c + c'$ to p for each constraint $x_i' - x_i \lesssim c'$ in p. (In terms of the constraint graph defined by p, relaxation with $x_i - x_j \lesssim c$ corresponding to an edge from x_j to x_i creates a new edge from x_j to x_i' with weight $c + c'$ for each edge from x_i to x_i' with weight c' (i.e., the edge from x_j to x_i' is now explicit, not implicit via x_i).) The case where x is equal to x_j is symmetric. These observations lead to a function for performing relaxation shown in Algorithm 17.

The well-known technique of dynamic programming is preferably used in both EXISTS and RELAX to increase efficiency.

Universal quantification

Universal and existential quantification are related through the identity $\forall x.\phi = \neg\exists x.\neg\phi$. Hence, a universal quantification algorithm can be expressed in terms of NOT and EXISTS as shown in Algorithm 18.

Manipulators

The language of difference constraint expressions can be extended with two useful operations: assignment and replacement:

$$\phi' \quad ::= \quad \phi \mid \phi[x \leftarrow y + c] \mid \phi[y + c \,/\, x]$$

An *assignment* gives a variable x the value of a variable y plus a constant c:

$$[\![\phi[x \leftarrow y + c]]\!]a \quad \stackrel{\text{def}}{=} \quad [\![\phi]\!]a[x \mapsto a(y) + c]. \tag{10}$$

For example, let $\phi_1 = (x - y = 1) \wedge (y - z = 1)$, and $a = [x \mapsto 5, y \mapsto 1, z \mapsto 0]$. Clearly, $[\![\phi_1]\!]a = \textbf{false}$. We now perform the assignment $x \leftarrow z + 2$ in ϕ_1, and get:

$$\begin{aligned} [\![\phi_1[x \leftarrow z + 2]]\!]a &= [\![\phi_1]\!]a[x \mapsto a(z) + 2] \\ &= [\![\phi_1]\!][x \mapsto 2, y \mapsto 1, z \mapsto 0] \\ &= \textbf{true}. \end{aligned}$$

When $x \neq y$, performing an assignment corresponds to removing all explicit bounds on x, and then updating x with a new value. The assignment operation $\phi[x \leftarrow y + c]$ is therefore performed as:

$$\phi[x \leftarrow y + c] = (\exists x.\phi) \wedge (x - y = c) \qquad \text{if } x \neq y\,.$$

which gives the algorithm shown in Algorithm 19.

An assignment $x \leftarrow y + c$ in which $x = y$ corresponds to incrementing x by the value c. Because the upper bound is changed on all nodes comprising x and the variables are not rearranged in the DDD x is simply incremented, INCREMENT shown in Algorithm 20 can recursively create a new DDD with MK. Again the well-known technique of dynamic programming is preferably used in INCREMENT to make it efficient.

The *replacement* operator is closely related to assignment. A replacement $\phi[y + c/x]$ syntactically substitutes all occurrences of x in ϕ with a variable y plus a constant c. When the two variables are different, a replacement is performed as:

$$[\![\phi[y + c/x]]\!] \stackrel{\text{def}}{=} [\![\exists x.((x - y = c) \wedge \phi)]\!].$$

If x is equal to y, the replacement $\phi[x + d/x]$ is defined as $\phi[t/x][x + d/t]$, where t is a variable different from x and not occurring in ϕ.

It can be observed that for a terminal node the result is the terminal and that for a non-terminal node, the replacement can be performed syntactically on the attributes of u. For example, substituting x_1 by $x_2 + c_2$ in (x_3, x_1, o, c_1) gives $(x_3, x_2, o, c_1 + c_2)$. These observations yield the function shown in Algorithm 21. Again the use of dynamic programming in REPLACE(u, x, y, c) makes it efficient, and only a linear number of new constraints are constructed. However, in order to maintain orderedness these new constraints cannot be added where they are discovered through calls to MK, but are added through calls to APPLY.

Convex Hull

The smallest convex set expressible by a difference constraint expression, called the convex hull, can for example be computed by enumerating all 1-paths, running Floyd-Warshall on each of these paths, and finally combining them into one matrix by element-wise taking the greatest entry in the matrices. This is done by the function HULL shown Algorithm 22.

Disjunctive Nodes

Let p be a path leading to the node u in a DDD, and assume $\alpha = cstr(u)$, $h = high(u)$, and $l = low(u)$. Then u is *disjunctive in* p if $[p] \wedge (\alpha \rightarrow h, l)$ and $[p] \wedge (h \vee l)$ are equivalent. (Here, $[p]$ denotes the system of difference constraints induced by a path p). If a node is disjunctive in a path, the node can be omitted from the path. A function for removing all disjunctive nodes in a DDD is shown in Algorithm 23.

Generalizations

In situations where the invention is used to analyze systems which require the use of linear inequalities, the preferred embodiment is as described above for difference constraints with a minor modification. Instead of using a shortest path algorithm such as Bellman-Ford to check for the feasibility of paths in DDDs, an algorithm such as Simplex for solving linear programming is preferably used.

A *linear inequality expression* can be expressed in the following syntax:

$$\phi ::= \sum_{i=1}^{n} a_i y_i \leq b \mid \phi \vee \phi \mid \neg\phi \mid \exists y_i.\phi \,.$$

The nodes of the data strucutre comprise expressions of the form $\sum_{i=1}^{n} a_i y_i \leq b$, which, as usual, is a shorthand for the expression $a_1 y_1 + \cdots + a_n y_n \leq b$. Here, a_i is an integer- or real-valued constant, and y_i is an integer- or real-valued variable ($i = 1, \ldots, n$). The expressions of the nodes are ordered according to a predetermined criteria.

The basic algorithms for constructing nodes are adapted to linear inequalities in a straightforward manner. The algorithm for combining two data structures is only changed such that the correct ordering is obtained by comparing the order of the linear inequality expressions (and not only the two variables and the bound) comprised in the nodes. The algorithm for performing feasibility check is substituted by algorithms for solving linear programming problems or integer linear programming problems. The algorithms for determining functional properties are straightforward to adapt using the feasibility checking algorithm. The algorithm for finding a satisfying variable assignment can also be constructed in a straightforward manner using the algorithms for solving linear programming problems. An algorithm for performing existential quantification can be obtained using Fourier-Motzkin variable elimination along the lines of Algorithm 16 for existential quantification for difference decision diagrams.

Annex A: Algorithms

Algorithm 1

function MK$((x,y),(o,c),h,l)$: Pair $\times$ Bound $\times V \times V \to V$
  if $\mathbb{D} = \mathbb{Z} \wedge o =$ LE **then**
    $(o,c) \leftarrow$ (LEQ, $c-1$)
  if $member(G,(x,y,o,c,h,l))$ **then**
    return $lookup(G,(x,y,o,c,h,l))$
  elsif $l = h$ **then**
    return l
  elsif $(x,y) = var(l) \wedge h = high(l)$ **then**
    return l
  else
    return $insert(G,(x,y,o,c,h,l))$

Algorithm 2

function MKNORM$((x,y),b,h,l)$: Pair $\times$ Bound $\times V \times V \to V$
  if $x = y$ **then**
    if $b <$ (LEQ, 0) **then**
      return 0
    else
      return 1
  elsif $x > y$ **then**
    return MK$((x,y),b,h,l)$
  else
    return MK$((y,x),-b,l,h)$

Algorithm 3

function MKDIFFCSTR$(x, y, o, c) : \text{Var} \times \text{Var} \times \{\text{EQ}, \text{NEQ}, \text{LEQ}, \text{GEQ}, \text{LE}, \text{GR}\} \times \mathbb{D} \to V$
  if $o = \text{LE}$ **then** ▷ $x - y < c$
    return MKNORM$((x, y), (\text{LE}, c), \mathbf{1}, \mathbf{0})$
  elsif $o = \text{LEQ}$ **then** ▷ $x - y \leq c$
    return MKNORM$((x, y), (\text{LEQ}, c), \mathbf{1}, \mathbf{0})$
  elsif $o = \text{EQ}$ **then then** ▷ $x - y = c$
    return MKNORM$\big((x, y), (\text{LE}, c), \mathbf{0}, \text{MKNORM}((x, y), (\text{LEQ}, c), \mathbf{1}, \mathbf{0})\big)$
  elsif $o = \text{NEQ}$ **then** ▷ $x - y \neq c$
    return MKNORM$\big((x, y), (\text{LE}, c), \mathbf{1}, \text{MKNORM}((x, y), (\text{LEQ}, c), \mathbf{0}, \mathbf{1})\big)$
  elsif $o = \text{GR}$ **then** ▷ $x - y > c$
    return MKNORM$((x, y), (\text{LEQ}, c), \mathbf{0}, \mathbf{1})$
  else ▷ $x - y \geq c$
    return MKNORM$((x, y), (\text{LE}, c), \mathbf{0}, \mathbf{1})$

Algorithm 4

function APPLY$(op, u, v) : \text{Conn} \times V \times V \to V$
  if $u, v \in \{\mathbf{0}, \mathbf{1}\}$ **then**
    return $eval(op, u, v)$
  elsif $member(H, (op, u, v))$ **then**
    return $lookup(H, (op, u, v))$
  elsif $var(u) < var(v)$ **then**
    $r \leftarrow \text{MK}\big(var(u), bound(u), \text{APPLY}(op, high(u), v), \text{APPLY}(op, low(u), v)\big)$
  elsif $var(u) = var(v) \wedge bound(u) < bound(v)$ **then**
    $r \leftarrow \text{MK}\big(var(u), bound(u), \text{APPLY}(op, high(u), high(v)), \text{APPLY}(op, low(u), v)\big)$
  elsif $var(u) = var(v) \wedge bound(u) = bound(v)$ **then**
    $r \leftarrow \text{MK}\big(var(u), bound(u), \text{APPLY}(op, high(u), high(v)), \text{APPLY}(op, low(u), low(v))\big)$
  elsif $var(u) = var(v) \wedge bound(u) > bound(v)$ **then**
    $r \leftarrow \text{MK}\big(var(v), bound(v), \text{APPLY}(op, high(u), high(v)), \text{APPLY}(op, u, low(v))\big)$
  else
    $r \leftarrow \text{MK}\big(var(v), bound(v), \text{APPLY}(op, u, high(v)), \text{APPLY}(op, u, low(v))\big)$
  $insert(H, (op, u, v), r)$
  return r

Algorithm 5

```
function NOT(u) : V → V
  if u = 0 then
    return 1
  elsif u = 1 then
    return 0
  elsif member(H, (NP1, u, 0)) then
    return lookup(H, (NP1, u, 0))
  else
    r ← MK(var(u), bound(u), NOT(high(u)), NOT(low(u)))
  insert(H, (NP1, u, 0), r)
  return r
```

Algorithm 6

```
function FEASIBLE(M) : Bound-matrix → 𝔹
  FLOYD-WARSHALL(M)
  for i ← 1 to n do
    if M_ii < (LEQ, 0) then
      return false
  return true
```

Algorithm 7

```
function REDUCE(u) : V → V
  return reduce(u, {}, Array(0,0), ⟨⟩)
where
  function reduce(v, W, d, L)
    if ¬FEASIBLE'(|W|, d, L) then
      return ⊥
    elsif v ∈ {0,1} then
      return v
    else
      (x_i, x_j) ← var(v)
      d^h ← d^l ← d
      L^h ← L^l ← L
      W' ← W ∪ {x_i, x_j}
      INSERTCSTR(L^h, ((x_j, x_i), bound(v)))
      INSERTCSTR(L^l, ((x_i, x_j), −bound(v)))
      h ← reduce(high(v), W', d^h, L^h)
      l ← reduce(low(v), W', d^l, L^l)
      if l ≠ ⊥ ∧ h ≠ ⊥ then
        return MK(var(v), bound(v), h, l)
      elsif h ≠ ⊥ then
        return h
      else
        return l
```

Algorithm 8

```
function FEASIBLE'(m, d, L) : ℤ × Bound-array × Cstr-list → 𝔹
  i ← 0
  repeat
    unchanged ← relax(d, L)
    i ← i + 1
  until i > m ∨ unchanged
  return unchanged
where
  function relax(d, L)
    unchanged' ← true
    foreach ((x_i, x_j), b) ∈ L do        ▷ b is the distance from x_i to x_j
      if d_j ≻ b + d_i then               ▷ d_j is the distance from x_0 to x_j
        d_j ← b + d_i
        unchanged' ← false
    return unchanged'
```

Algorithm 9

```
function INSERTCSTR(L, (v, b))
  if L = ⟨⟩ then
    L ← ⟨(v, b)⟩
  else
    (v', b') ← head L
    if v' = v then
      L ← (v, b)^tail L
    else
      L ← (v, b)^L
```

Algorithm 10

```
function UNSATISFIABLE(u) : V
  return REDUCE(u) = 0

function TAUTOLOGY(u) : V
  return REDUCE(u) = 1

function SATISFIABLE(u) : V
  return REDUCE(u) ≠ 0

function FALSIFIABLE(u) : V
  return REDUCE(u) ≠ 1

function EQUIVALENT(u, v) : V × V
  return TAUTOLOGY(APPLY(BIIMP, u, v))

function CONSEQUENCE(u, v) : V × V
  return TAUTOLOGY(APPLY(IMP, u, v))
```

Algorithm 11

function ALLINFEASIBLE$(v, t, W, d, L) : V \times V \times$ Var-set $\times$ Bound-array $\times$ Cstr-list $\to \mathbb{B}$
  $ok \leftarrow$ FEASIBLE$'(|W|, d, L)$
  if $v \in \{0, 1\} \vee \neg ok$ **then**
    return $\neg(v = t \wedge ok)$
  else
    $(x_i, x_j) \leftarrow var(v)$
    $L^h \leftarrow L^l \leftarrow L$
    $d^h \leftarrow d^l \leftarrow d$
    INSERTCSTR$(L^h, ((x_j, x_i), bound(v)))$
    INSERTCSTR$(L^l, ((x_i, x_j), -bound(v)))$
    $W' \leftarrow W \cup \{x_i, x_j\}$
    if ALLINFEASIBLE$(high(v), t, W', d^h, L^h)$ **then**
      return ALLINFEASIBLE$(low(v), t, W', d^l, L^l)$
    else
      return false

Algorithm 12

function EXISTSFEASIBLE$(v, t, W, d, L) : V \times V \times$ Var-set $\times$ Bound-array $\times$ Cstr-list $\to \mathbb{B}$
  $ok \leftarrow$ FEASIBLE$'(|W|, d, L)$
  if $v \in \{0, 1\} \vee \neg ok$ **then**
    return $v = t \wedge ok$
  else
    $(x_i, x_j) \leftarrow var(v)$
    $L^h \leftarrow L^l \leftarrow L$
    $d^h \leftarrow d^l \leftarrow d$
    INSERTCSTR$(L^h, ((x_j, x_i), bound(v)))$
    INSERTCSTR$(L^l, ((x_i, x_j), -bound(v)))$
    $W' \leftarrow W \cup \{x_i, x_j\}$
    if $\neg$EXISTSFEASIBLE$(high(v), t, W', d^h, L^h)$ **then**
      return EXISTSFEASIBLE$(low(v), t, W', d^l, L^l)$
    else
      return true

Algorithm 14

```
function UNSATISFIABLE(u) : V
  return ALLINFEASIBLE(u, 1, {}, Array(0,0), ⟨⟩)

function TAUTOLOGY(u) : V
  return ALLINFEASIBLE(u, 0, {}, Array(0,0), ⟨⟩)

function SATISFIABLE(u) : V
  return EXISTSFEASIBLE(u, 1, {}, Array(0,0), ⟨⟩)

function FALSIFIABLE(u) : V
  return EXISTSFEASIBLE(u, 0, {}, Array(0,0), ⟨⟩)
```

Algorithm 15

```
function ANYSAT(u) : V → Asgn ∪ {⊥}
  return anysat(u, {}, Array(0,0), ⟨⟩)
where
  function anysat(v, W, d, L)
    ok ← FEASIBLE'(|W|, d, L)
    if v = 1 ∧ ok then
      for i ← 1 to n do
          (δ, c) ← d_i
          if δ = 0 then
            a ← a[x_i ↦ c]
          else
            a ← a[x_i ↦ c − ε]
      return a
    elsif v ∈ {0, 1} ∨ ¬ok then
      return ⊥
    else
      (x_i, x_j) ← var(v)
      L^h ← L^l ← L
      d^h ← d^l ← d
      INSERTCSTR(L^h, ((x_j, x_i), bound(v)))
      INSERTCSTR(L^l, ((x_i, x_j), −bound(v)))
      W' ← W ∪ {x_i, x_j}
      a ← anysat(high(v), W', d^h, L^h)
      if a = ⊥ then
        return anysat(low(v), W', d^l, L^l)
      else
        return a
```

Algorithm 16

function EXISTS(x, u) : Var $\times V \to V$
  if $u \in \{0, 1\}$ **then return** u
  elsif $x \in \{pos(u), neg(u)\}$ **then**
    $h \leftarrow$ RELAX$(high(u), x, pos(u), neg(u), bound(u))$
    $l \leftarrow$ RELAX$(low(u), x, neg(u), pos(u), -bound(u))$
    return EXISTS$(x, h) \vee$ EXISTS(x, l)
  else
    $\alpha \leftarrow$ MK$(var(u), bound(u), \mathbf{1}, \mathbf{0})$
    $h \leftarrow$ EXISTS$(x, high(u))$
    $l \leftarrow$ EXISTS$(x, low(u))$
    return $(\alpha \wedge h) \vee (\neg\alpha \wedge l)$

Algorithm 17

function RELAX(u, x, x_i, x_j, b) : $V \times$ Var $\times$ Var $\times$ Var $\times$ Bound $\to V$
  if $u \in \{0, 1\}$ **then return** u
  else
    $h \leftarrow$ RELAX$(high(u), x, x_i, x_j, b)$
    $l \leftarrow$ RELAX$(low(u), x, x_i, x_j, b)$
    $\alpha \leftarrow$ MK$(var(u), bound(u), \mathbf{1}, \mathbf{0})$
    if $neg(u) = x_i = x$ **then**
      $h \leftarrow h \wedge$ MKNORM$\big((pos(u), x_j), b + bound(u), \mathbf{1}, \mathbf{0}\big)$
    elsif $pos(u) = x_i = x$ **then**
      $l \leftarrow l \wedge$ MKNORM$\big((neg(u), x_j), b - bound(u), \mathbf{1}, \mathbf{0}\big)$
    elsif $neg(u) = x_j = x$ **then**
      $l \leftarrow l \wedge$ MKNORM$\big((x_i, pos(u)), b - bound(u), \mathbf{1}, \mathbf{0}\big)$
    elsif $pos(u) = x_j = x$ **then**
      $h \leftarrow h \wedge$ MKNORM$\big((x_i, neg(u)), b + bound(u), \mathbf{1}, \mathbf{0}\big)$
    return $(\alpha \wedge h) \vee (\neg\alpha \wedge l)$

Algorithm 18

function FORALL(x, u) : Var $\times V \to V$
  return NOT(EXISTS$(x,$ NOT$(u))$)

Algorithm 19

function ASSIGN$(u, x, y, c) : V \times$ Var $\times$ Var $\times \mathbb{D} \to V$
  if $x = y$ **then**
    return INCREMENT(u, x, c)
  else
    return APPLY(AND, MKDIFFCSTR$(x, y,$ EQ$, c)$, EXISTS(x, u))

Algorithm 20

function INCREMENT$(u, x, c) : V \times$ Var $\times \mathbb{D} \to V$
  if $u \in \{0, 1\}$ **then**
    return u
  else
    if $pos(u) = x$ **then**
      $b \leftarrow bound(u) +$ (LEQ$, c$)
    elsif $neg(u) = x$ **then**
      $b \leftarrow bound(u) +$ (LEQ$, -c$)
    return MK$(var(u), b,$ INCREMENT$(high(u), x, c),$ INCREMENT$(low(u), x, c))$

Algorithm 21

function REPLACE$(u, x, y, c) : V \times$ Var $\times$ Var $\times \mathbb{D} \to V$
  if $u \in \{0, 1\}$ **then**
    return u
  else
    if $pos(u) = x$ **then**
      $u' \leftarrow$ MKNORM$((y, neg(u)), bound(u) -$ (LEQ$, c), \mathbf{1}, \mathbf{0})$
    elsif $neg(u) = x$ **then**
      $u' \leftarrow$ MKNORM$((pos(u), y), bound(u) +$ (LEQ$, c), \mathbf{1}, \mathbf{0})$
    else
      $u' \leftarrow$ MK$(var(u), bound(u), \mathbf{1}, \mathbf{0})$
    $h \leftarrow$ REPLACE$(high(u), x, y, c)$
    $l \leftarrow$ REPLACE$(low(u), x, y, c)$
    return APPLY(OR, APPLY(AND$, u', h)$, APPLY(AND, NOT$(u'), l)$)

Algorithm 22

```
function HULL(u) : V → Bound-matrix
  M ← Matrix(LE, ∞)
  hull(u, M)
  return M
where
  function hull(v, M)
    if v = 0 then
      M ← ⊥
    elsif v = 1 then
      FLOYD-WARSHALL(M)
    else
      (x_i, x_j) ← var(v)
      M^h ← M^l ← M
      M^h_ji ← bound(v)
      M^l_ij ← −bound(v)
      hull(high(v), M^h)
      hull(low(v), M^l)
      if M^h ≠ ⊥ ∧ M^l ≠ ⊥ then
        M ← MAX(M^h, M^l)
      elsif h ≠ ⊥ then
        M ← M^h
      elsif l ≠ ⊥ then
        M ← M^l
      else
        M ← ⊥
```

Algorithm 23

```
function MERGE(u) : V → V
  return merge(u, {}, Array(0,0), ⟨⟩)
where
  function merge(v, W, d, L)
    if v ∈ {0,1} then
      return v
    else
      (x_i, x_j) ← var(v)
      L^h ← L^l ← L
      d^h ← d^l ← d
      W' ← W ∪ {x_i, x_j}
      INSERTCSTR(L^h, ((x_j, x_i), bound(v)))
      INSERTCSTR(L^l, ((x_i, x_j), −bound(v)))
      h ← merge(high(v), W', d^h, L^h)
      l ← merge(low(v), W', D^l, L^l)
      if ALLINFEASIBLE((h ∨ l) ↔ v, 0, W, d, L) then
        return h ∨ l                          ▷ v is disjunctive
      else
        return MK(var(v), bound(v), h, l)
```

Claims

1. An acyclic data structure comprising:

   - a number of nodes comprising
     - at least a first and a second pointer pointing to other nodes,
     - an expression comprising at least one inequality with at least one variable, the expression being adapted to result in one of at least two disjoint outcomes, each pointer representing one of the outcomes, the number of pointers corresponding to the number of outcomes of the expression,
   - at least one terminal node,
   - at least one node pointing to the at least one terminal node,

   the expressions being ordered according to predetermined criteria, the pointers of a first node comprising an expression of a first, lower order pointing to nodes comprising expressions of second orders, the second orders being higher than the first order.

2. A data structure according to claim 1, wherein the data structure is at least substantially free from incidents of nodes where:

   - the first and second pointers of a first node point to a second and a third node, respectively,
   - the second pointer of the second node points to the third node,
   - the expressions of the first and second nodes relate to the same variables, and
   - the variable values fulfilling or not fulfilling the expression of the first node being comprised in the variable values fulfilling or not fulfilling the expression of the second node.

3. A data structure according to claim 1 or 2, wherein the data structure is at least substantially free from incidents of nodes where all pointers of a node point to the same node.

4. A data structure according to any of the preceding claims, wherein the data structure is at least substantially free from incidents of nodes where two nodes exist having identical expressions and having pointers pointing to the same nodes, where the first pointers of the two nodes point to the same node, and where the second pointers of the two nodes point to the same node.

5. A data structure according to any of the preceding claims, wherein the terminal nodes are adapted to represent Boolean values "true" and "false".

6. A data structure according to any of the preceding claims, wherein the expressions in the nodes except the terminal nodes all contain at least one inequality.

7. A data structure according to any of the preceding claims, wherein the disjoint outcomes of the expressions constitute "true" or "false", and wherein each node comprises two pointers.

8. A data structure according to claim 6 or 7, wherein the at least one inequality is a linear inequality.

9. A data structure according to claim 6, 7, or 8, wherein the inequalities are difference constraints.

10. A data structure according to any of the preceding claims, wherein the data structure is at least substantially free from incidents of nodes where, when following a path from one node via one or more pointers to a second node, there exists no set of variable values fulfilling a combined expression obtained by, for each node entered, the expression therein having to provide the outcome corresponding to the pointer of the node pointing to the next node.

11. A data structure according to any of the preceding claims, wherein the data structure is at least substantially free from incidents of pairs of paths, starting in the same starting node and ending in the same ending node, where a single path may be generated starting in the starting node and ending in the ending node, so that the same set of variable values fulfill the combined expression obtained when following the single path from the starting node to the ending node as fulfill a disjunction of the pair of paths.

12. A method of generating a data structure according to claim 1 and representing a system having a number of variables, the method comprising:

a) determining the variables,

b) defining a number of entities in the system, the entities defining relations between variables,

c) defining criteria for ordering the expressions,

d) representing each relation by:

- defining a number of different expressions each comprising at least one inequality with at least one variable, and each expression being adapted to result in one of at least two disjoint outcomes,
- generating a node associated to each expression, the node having: at least a first and a second pointer pointing to other nodes, the number of pointers of the node corresponding to the number of outcomes of the expressions,

- ordering the expressions associated with the nodes in accordance with the defined criteria so that the pointers of a node comprising an expression of a lower order points to nodes comprising expressions of higher orders so as to generate an entity data structure representing the corresponding entity,

e) combining the entity data structures to generate the data structure.

13. A method according to claim **12** further comprising:

a) on the basis of the combined data structure determining at least one functional property of the system.

14. A method according to claim **12**, wherein step e) comprises a number of steps in each of which a number of entity data structures are combined, each step comprising:

a) in the system determining a relationship between the entities represented by the entity data structures and a mathematical operation determined by the relationship,

b) generating a new data structure by generating an operator node representing the mathematical operation and having a number of pointers pointing to the entity data structures.

15. A method according to claim **3**, wherein:

- a first node is identified, all pointers of which point to the same, second node
- all pointers pointing to the first node are pointed to the second node, and
- the first node is removed.

16. A method according to claim **3** or **4**, wherein:

- two nodes are identified having identical expressions and having pointers pointing to the same nodes, where the first pointers of the two nodes point to the same node, and where the second pointers of the two nodes point to the same node,
- pointing all pointers pointing to a first of the two nodes to the other of the two nodes, and deleting the first node.

17. A method according to claim **14**, wherein a set of predetermined reduction rules are repeatedly applied to the operator nodes in order to remove operator nodes from the data structure.

18. A method according to any of claims **14**, **15**, **16** and **17**, further comprising the step of:

- identifying an operator node having pointers pointing to more than two data structures,

- replacing the identified operator node by a group of operator nodes, each operator node in the group having two pointers, the group of operator nodes pointing to the more than two data structures.

19. A method according to claim 18, further comprising the step of:

a) identifying an operator node having pointers pointing to two data structures comprising only terminal nodes or nodes the expressions of which represent inequalities,

b) replacing the identified operator node and the data structures pointed to thereby by a new data structure generated by performing the following procedure relating to the two data structures:

c)
- if the lowest order node of the first data structure and the lowest order node of the second data structure comprise identical expressions,
  - generating a new node having an expression identical thereto,
  - generating a first new data structure from the data structures pointed to by the first pointers of the two lowest order nodes by performing step c),
  - having the new node's first pointer point at the first new data structure,
  - generating a second new data structure from the data structures pointed to by the second pointers of the two lowest order nodes by performing step c),
  - having the new node's second pointer point at the second new data structure,
- if the lowest order node of the first data structure and the lowest order node of the second data structure comprise different expressions,
  - generating a new node having an expression identical to that of the two nodes having the lowest order,
  - generating a first new data structure from the data structures pointed to by the first pointer of the node having the lowest order and that node not having the lowest order by performing step c),
  - having the new node's first pointer point at the first new data structure,
  - generating a second new data structure from the data structures pointed to by the second pointer of the node having the lowest order and that node not having the lowest order by performing step c),
  - having the new node's second pointer point at the second new data structure,
- if the lowest order node of one of the data structures comprises an expression, and the other data structure is a terminal node,
  - generating a new node having an expression identical to that of the node comprising an expression,

- generating a first new data structure from the data structures pointed to by the first pointer of the node comprising an expression and the terminal node by performing step c),
- having the new node's first pointer point at the new data structure,
- generating a second new data structure from the data structures pointed to by the second pointer of the node comprising an expression and the terminal by performing step c),
- having the new node's second pointer point at the second new data structure,

• if the two data structures are terminal nodes, performing the mathematical operation of the operator node between the terminal nodes and generating a data structure consisting of a terminal node representing the result of the operation.

20. A method according to claim 12, wherein the combination of the entity data structures comprises:

a) in the system determining a relationship between the two entities represented by the two data structures and a mathematical operation determined by the relationship,

b) generating a new data structure by performing the following procedure relating the two data structures:

c) • if the lowest order node of the first data structure and the lowest order node of the second data structure comprise identical expressions,
  - generating a new node having an expression identical thereto,
  - generating a first new data structure from the data structures pointed to by the first pointers of the two lowest order nodes by performing step c),
  - having the new node's first pointer point at the first new data structure,
  - generating a second new data structure from the data structures pointed to by the second pointers of the two lowest order nodes by performing step c),
  - having the new node's second pointer point at the second new data structure,

• if the lowest order node of the first data structure and the lowest order node of the second data structure comprise different expressions,
  - generating a new node having an expression identical to that of the two nodes having the lowest order,
  - generating a first new data structure from the data structures pointed to by the first pointer of the node having the lowest order and that node not having the lowest order by performing step c),

- having the new node's first pointer point at the first new data structure,
- generating a second new data structure from the data structures pointed to by the second pointer of the node having the lowest order and that node not having the lowest order by performing step c),
- having the new node's second pointer point at the second new data structure,

• if the lowest order node of one of the data structures comprises an expression, and the other data structure is a terminal node,

- generating a new node having an expression identical to that of the node comprising an expression,
- generating a first new data structure from the data structures pointed to by the first pointer of the node comprising an expression, and the terminal node by performing step c),
- having the new node's first pointer point at the new data structure,
- generating a second new data structure from the data structures pointed to by the second pointer of the node comprising an expression and the terminal by performing step c),
- having the new node's second pointer point at the second new data structure,

• if the two data structures are terminal nodes, performing the mathematical operation between the terminal nodes and generating a data structure consisting of a terminal node representing the result of the operation,

d) repeating steps a) and b) until only a single data structure remains.

21. A method of generating a new data structure according to claim **1** by combining two such data structures using a mathematical operation, the method comprising:

a) generating the new data structure by:

• if the lowest order node of the first data structure and the lowest order node of the second data structure comprise identical expressions,

- generating a new node having an expression identical thereto,
- generating a first new data structure from the data structures pointed to by the first pointers of the two lowest order nodes by performing step a),
- having the new node's first pointer point at the first new data structure,
- generating a second new data structure from the data structures pointed to by the second pointers of the two lowest order nodes by performing step a),

- having the new node's second pointer point at the second new data structure,

- if the lowest order node of the first data structure and the lowest order node of the second data structure comprise different expressions,
  - generating a new node having an expression identical to that of the two nodes having the lowest order,
  - generating a first new data structure from the data structures pointed to by the first pointer of the node having the lowest order and that node not having the lowest order by performing step a),
  - having the new node's first pointer point at the first new data structure,
  - generating a second new data structure from the data structures pointed to by the second pointer of the node having the lowest order and that node not having the lowest order by performing step a),
  - having the new node's second pointer point at the second new data structure,
- if the lowest order node of one of the data structures comprises an expression, and the other data structure is a terminal node,
  - generating a new node having an expression identical to that of the node comprising an expression,
  - generating a first new data structure from the data structures pointed to by the first pointer of the node comprising an expression and the terminal node by performing step a),
  - having the new node's first pointer point at the new data structure,
  - generating a second new data structure from the data structures pointed to by the second pointer of the, node comprising an expression and the terminal by performing step a),
  - having the new node's second pointer point at the second new data structure,
- if the two data structures are terminal nodes, performing the mathematical operation between the terminal nodes and generating a data structure consisting of a terminal node representing the result of the operation.

22. A method according to any of claims 14–20, wherein the mathematical operations are chosen from the group consisting of Boolean operators or combinators, such as AND, OR, NOT, and XOR, where the terminal nodes are given one of the values "true" and "false".

23. A method according to claim 21, wherein the mathematical operation is chosen from the group

consisting of Boolean operators or combinators, such as AND, OR, NOT, and XOR, where the terminal nodes are given one of the values "true" and "false".

24. A method according to claim 22, wherein the mathematical operations are binary operations, and where the nodes comprising expressions are generated with a first and a second pointer so as to be able to point at two other nodes, the second pointer being used, if the expression, given a set of variable values, is true, and the first pointer if the expression is false.

25. A method according to claim 24, wherein:

- a first node is identified, all pointers of which point to the same, second node,
- all pointers pointing to the first node are pointed to the second node, and
- the first node is removed.

26. A method according to claim 24 or 25, wherein:

- two nodes are identified having identical expressions and having pointers pointing to the same nodes, where the first pointers of the two nodes point to the same node, and where the second pointers of the two nodes point to the same node,
- pointing all pointers pointing to a first of the two nodes to the other of the two nodes, and
- deleting the first node.

27. A method according to any of claims 24–26, wherein

- three nodes are identified where:
  - the first and second pointers of a first node point to a second and a third node, respectively,
  - the second pointer of the second node points to the third node,
  - the expressions of the first and second nodes relate to the same variables, and the variable values fulfilling the expression of the first node being comprised in the variable values fulfilling the expression of the second node, and
- replacing pointers pointing to the first node by pointers pointing to the second node.

28. A method according to any of claims 24–27, wherein

- three nodes are identified where:
  - the second and first pointers of a first node point to a second and a third node, respectively,

- the first pointer of the second node points to the third node,
- the expressions of the first and second nodes relate to the same variables, and the variable values not fulfilling the expression of the first node being comprised in the variable values not fulfilling the expression of the second node, and

• pointing the first node's second pointer to a node pointed to by the second node's second pointer.

29. A method according to claim 27 or 28, wherein the second node is subsequently removed, if no pointers point to it.

30. A method according to any of claims 12–29, wherein, during step e) or during generation of an entity data structure, if a new node is to be generated having all pointers point to the same, second node, where one or more pointers were to point to the new node, not inserting the new node and directing all pointers pointing to the new node to the second node.

31. A method according to any of claims 12–30, wherein, during step e) or during generation of an entity data structure, if a new node is to be generated having an expression identical to that of a second node, a first pointer pointing to a node pointed to by a first pointer of the second node, and a second pointer pointing to a node pointed to by a second pointer of the second node, where one or more pointers were to point to the new node, not inserting the new node and directing all pointers pointing to the new node to the second node.

32. A method of altering a data structure according to claim 5 and 9, wherein

• identifying all paths leading from a root to a "true" terminal node,

• for each path, constructing a difference bound matrix obtained from a combined expression obtained by, for each node entered in the path, the expression therein having to provide the outcome corresponding to the pointer of the node pointing to the next node,

• solving the all pairs shortest path problem for each difference bound matrix,

• removing in each matrix the row and column corresponding to a predetermined variable,

• constructing a path from each matrix, and

• combining all the paths by a disjunction using the method of claim 23.

33. A method according to claim 32, wherein the construction of a path from each matrix comprises, for each entry in the matrix, generating a node having a difference constraint corresponding to the variables of the row and column and the constant of the entry, and subsequently combining the resulting nodes by conjunction.

34. A method according to claim 32 or 33, wherein the solving step comprises, for each matrix, solving the difference bound matrix by the algorithm of Floyd-Warshall performing only relaxation steps involving the predetermined variable.

35. A method of altering a data structure according to claim 5 and either 8 or 9, the method comprising:

(a) determining a variable,

(b) determining a constraining expression which is either a lower or an upper bound on the variable,

(c) generating a new data structure by:

- if the data structure is a terminal node then the result is said terminal node,
- if the node of the data structure having the lowest order comprises an expression containing the variable,
  - generating a first new data structure from the data structure pointed to by the first pointer of the node by performing step c),
  - generating a second new data structure from the data structure pointed to by the second pointer of the node by performing step c),
  - if the constraining expression is an upper bound on the variable and the expression of the node is also an upper bound on the variable,
    * constructing a new expression without the variable obtained by combining conjunctively the constraining expression and the negation of the expression of the node,
    * generating the resulting data structure as the disjunction of
      · the negation of the expression of the node conjuncted with the first new data structure and the new, and
      · the expression of the node conjuncted with the second new data structure,
  - if the constraining expression is an upper bound on the variable and the expression of the node is a lower bound on the variable,
    * constructing a new expression without the variable obtained by combining conjunctively the constraining expression and the expression of the node,
    * generating the resulting data structure as the disjunction of
      · the negation of the expression of the node conjuncted with the first new data structure, and

· the expression of the node conjuncted with the second new data structure and the new expression,

– if the constraining expression is a lower bound on the variable and the expression of the node is an upper bound on the variable,

* constructing a new expression without the variable obtained by combining conjunctively the constraining expression and the expression of the node,

* generating the resulting data structure as the disjunction of

· the negation of the expression of the node conjuncted with the first new data structure, and

· the expression of the node conjuncted with the second new data structure and the new expression,

– if the constraining expression is a lower bound on the variable and the expression of the node is also a lower bound on the variable,

* constructing a new expression without the variable obtained by combining conjunctively the constraining expression and the negation of the expression of the node,

* generating the resulting data structure as the disjunction of

· the negation of the expression of the node conjuncted with the first new data structure and the new expression, and

· the expression of the node conjuncted with the second new data structure,

• if the node of the data structure having the lowest order does not comprise an expression containing the variable,

– generating a first new data structure from the data structure pointed to by the first pointer of the node by performing step c),

– generating a second new data structure from the data structure pointed to by the second pointer of the node by performing step c),

– generating a new node having an expression identical to the expression of the node,

– having the new node's first pointer point at the first new data structure,

– having the new node's second pointer point at the second new data structure.

36. A method of altering a data structure according to claim **5** and either **8** or **9**, the method comprising:

a) determining a variable,

b) generating a new data structure by:

- if the data structure is a terminal node then the result is said terminal node,
- if the node of the data structure having the lowest order does not comprise an expression containing the variable,
  - generating a first new data structure from the data structure pointed to by the first pointer of the node by performing step b),
  - generating a second new data structure from the data structure pointed to by the second pointer of the node by performing step b),
  - generating a new node having an expression identical to the expression of the node,
  - having the new node's first pointer point at the first new data structure,
  - having the new node's second pointer point at the second new data structure,
- if the node of the data structure having the lowest order comprises an expression containing the variable,
  - generating a first new data structure from the data structure pointed to by the first pointer of the node by performing the method according to claim 35 with the negation of the node's expression as the constraining expression and then performing step b),
  - generating a second new data structure from the data structure pointed to by the second pointer of the node by performing the method according to claim 35 with the node's expression as the constraining expression and then performing step b),
  - generating the resulting data structure as the disjunction of the first and the second new data structure.

37. A method for assessing whether, in a data structure according to any of claim 5 and 10, a set of variable values exists which, when starting in a root of the structure, would result in a path ending in a predetermined terminal node, the method comprising:

- inspecting whether the data structure consists of one terminal node only,
- if so, a positive answer is returned, if the only terminal node is the predetermined terminal node, and a negative answer is returned, if the only terminal node is not the predetermined terminal node,
- if not, a positive answer is returned.

38. A method for determining a set of variable values which, when starting in a predetermined root of a data structure according to claim 3, 5, and 10, results in a path ending in a predetermined terminal node, the method comprising:

- starting in the root of the structure and repeating the step of:

- if the first pointer of the node points to a terminal node different from the predetermined terminal node, selecting the node pointed to by the second pointer, otherwise selecting the node pointed to by the first pointer,

- if the predetermined terminal node is found:
  - constructing the path from the root to the terminal node and deriving a combined expression obtained by, for each node entered, the expression therein having to provide the outcome corresponding to the pointer of the node pointing to the next node, and
  - solving the combined expression and deriving a set of variable values of the solution.

39. A method for generating a result from a predetermined set of variable values when imposed on a data structure according to claim **1**, the method comprising:

- starting in a predetermined root of the structure and repeating the steps of:
  - if the node is a terminal node, returning the contents of the terminal node,
  - otherwise, evaluating the expression of the node according to the set of variable values and continuing with the node pointed at by the pointer corresponding to the outcome of the expression.

40. A method of altering a data structure according to claim **5** and **9**, the method comprising:

- interchanging the terminal nodes "true" and "false",
- removing the variable using the method according to any of claims **32–34** or **36**,
- interchanging the terminal nodes "true" and "false".

41. A method of altering a data structure according to claim **5** and **9**, the method comprising:

- replacing, in the data structure, a first variable x with the sum of a second, different variable y, and a constant c by:
- constructing a second data structure by conjugating the initial data structure with a data structure comprising a conjugation of a first node comprising a difference constraint relating to $x - y \leq c$, and a second node comprising a difference constraint relating to $x - y \geq c$,
- combining the first and the second data structures by the Boolean operation of conjunction using the method of claim **23**,
- removing x using the method of claim **32–34** or **36**.

42. A method of altering a data structure according to claim **5** and **9**, the method comprising replacing, in the data structure, a first variable x with the sum of a second, different variable y, and a constant c by:

- removing x from the data structure using the method of claim 32-34 or 36.
- constructing a second data structure by conjugating the initial data structure with a data structure comprising a conjugation of a first node comprising a difference constraint relating to $x - y \leq c$, and a second node comprising a difference constraint relating to $x - y \geq c$,
- combining the first and the second data structures by the Boolean operation of conjunction using the method of claim 23

43. A method of altering a data structure according to claim 5 and 9, the method comprising: in each expression comprising a predetermined variable, replacing the variable by the same variable added to a predetermined constant.

44. A method of obtaining information from a data structure according to claim 5 and 9, comprising the steps of:

- identifying all paths leading from a root to a "true" terminal node,
- for each path, constructing a difference bound matrix obtained from a combined expression obtained by, for each node entered in the path, the expression therein having to provide the outcome corresponding to the pointer of the node pointing to the next node,
- solving the all pairs shortest path problem for each difference bound matrix,
- generating a maximum matrix from the difference bound matrices and having the same dimensions as the difference bound matrices by, for each entry in the maximum matrix, selecting the largest value in the difference bound matrices relating to the same entry, and
- obtaining information from the maximum matrix.

45. A method for removing infeasible paths from a data structure according to claim 5 and 9, the method comprising, for each path in the data structure from a root node to a terminal node:

- for each node in the path, determining whether a set of variable values exists fulfilling a combined expression obtained by, for each node between the root node and the actual node, the expression therein having to provide the outcome corresponding to the pointer of the node pointing to the next node,
- removing the pointer in the path pointing to the actual node.

46. A method according to claim 45, wherein the determining step is performed according to the Bellman-Ford algorithm where, for each node in the path, information relating to the nodes already visited is stored and re-used in subsequent nodes.

47. A method for removing infeasible paths from a data structure according to claim 5 and 8, the method comprising, for each path in the data structure from a root node to a terminal node:

- for each node in the path, determining whether a set of variable values exists fulfilling a combined expression obtained by, for each node between the actual node and the root node, the expression therein having to provide the outcome corresponding to the pointer of the node pointing to the next node,
- removing the pointer in the path pointing to the actual node.

48. A method according to claim 47, wherein the determining step is performed using linear programming, such as the simplex algorithm, or using integer linear programming.

49. A method for altering a data structure according to claim 5, 7, and 9, the method comprising the steps of:

- identifying all paths leading from a root to a "true" terminal node,
- for each path, constructing a difference bound matrix obtained from a combined expression obtained by, for each node entered in the path, the expression therein having to provide the outcome corresponding to the pointer of the node pointing to the next node,
- solving the all pairs shortest path problem for each difference bound matrix,
- constructing a path from each matrix by expressing the bounds of each entry as difference constraints on the variables corresponding to the entry and forming the conjunction of the difference constraints, and
- generating an amended data structure by combining all the paths by a disjunction using the method of claim 23, and
- for each node in the amended data structure in each path from the root to a "true" terminal node:
  a) determining an initial expression from a combination of the expressions of the nodes in the path between the root and the actual node,
  b) determining a conjunctive combination between the initial expression and an expression obtained by a disjunction between the data structures pointed at by the two pointers of the node,
  c) determining a conjunctive combination between the initial expression and a disjunction between
     – a conjunction between the expression of the actual node and the data structure pointed at by the pointer representing a fulfillment of the expression of the node,

- a conjunction between the negation of the expression of the actual node and the data structure pointed at by the pointer representing a non-fulfillment of the expression of the node,

d) if the variable values fulfilling the combination b) and the combination c) are identical, replacing the actual node by the disjunction between the data structures pointed at by the two pointers of the actual node.

50. A method according to claim **12** for generating a data structure for analyzing a system modeled by a timed automaton having a number states and clocks, wherein:

step a) comprises:

- determining a first set of variables to be used for the encoding of the states,
- determining a second set of variables to be used for the clocks,

step b) comprises:

- identifying transitions between states, a transition comprising a starting state, an ending state, a requirement to be fulfilled in order to enable the transition to take place, an action to be performed when the transition takes place, and a requirement of the clocks to be fulfilled after the transition has taken place,

step d) comprises:

- for each transition, generating a data structure representing the requirement to be fulfilled in order for the transition to be enabled,

step e) comprises:

- constructing a data structure representing the set of reachable states by:
  - * constructing a data structure R representing a set of initial states of the automaton,
  - * repeatedly:
    - · selecting a transition,
    - · generating an amended data structure R' by conjugating the data structure representing the requirements of selected transition with R,
    - · generating an amended data structure R'' by, in R', updating variables in accordance with the actions of the transition,
    - · assigning R as the disjunction of R and R'',

  until R is unchanged for all transitions,

inquiring as to the existence of predetermined states of the automaton using any of the methods according to claim **37, 38, 39, and 44.**

51. A method according to claim 12 for generating a data structure for analyzing a concurrent system modeled by a composition of a number of timed automata each having a number of states and clocks, wherein:

step a) comprises:

- determining a first set of variables to be used for the encoding of the individual states of the automata,
- determining a second set of variables to be used for the individual clocks of the automata,
- determining a third and fourth set of variables to be used for encoding the new values of the variables from the first and second set such that there is a one-to-one correspondence between the variables in the first and third set, respectively in the second and fourth set,

step b) comprises:

- identifying non-idling transitions between states, a non-idling transition comprising a starting state, an ending state, a requirement to be fulfilled in order to enable the transition to take place, an action to be performed when the transition takes place, and a requirement of the clocks to be fulfilled after the transition has taken place,
- identifying idling transitions from a state to itself, comprising a requirement to be fulfilled when none of the requirements of the non-idling transitions are fulfilled on that state, an empty action, and a requirement of the clocks to be fulfilled after the transition has taken place,

step d) comprises:

- for each transition, generating a data structure over the four set of variables, representing a relation expressing the requirement to be fulfilled in order for the transition to be enabled using the first two set of variables, expressing the action to be performed when the transition takes place using the third and fourth set of variables, and expressing the requirement of the clocks using the third and fourth set of variables,
- generating a data structure A representing the advance time predicate using variables from the second and fourth set of variables,
- constructing a data structure T representing the set of transitions by:
  - * defining a data structure T as a terminal node representing "true",
  - * for each automaton:
    - · defining a data structure U as a terminal node representing "false",
    - · for each transition of the automaton, assigning to U the disjunction of U and the selected transition,

· assigning to T the conjunction of T and U,

* assigning to T the disjunction of the advance time predicate A and T,

step e) comprises:

– constructing a data structure representing the set of reachable states by:

* constructing a data structure R representing a set of initial states of the automata,

* repeatedly:

· generating a data structure R' by conjugating T and R,

· generating a data structure R'' by quantifying out all variables from the first and second set of variables using a method according to any of the claims **32–34** or **36**,

· generating a data structure R''' by replacing all variables from the third and fourth set of variables with the corresponding variable from the first and second set,

· assigning to R the disjunction of R and R''',

until R is unchanged,

inquiring as to the existence of predetermined states of the automata using a method according to any of the claims **37**, **38**, **39**, and **44**.

52. A method according to claim **12** for generating a data structure for analyzing a concurrent system modeled by a timed Petri net, the Petri net having a number of transitions and states, each state having a clock and an associated time delay interval, wherein:

step a) comprises:

– determining a first set of variables to be used for the encoding of the states,

– determining a second set of variables to be used for the clocks,

step b) comprises:

– identifying transitions between states, a transition comprising a starting state, an ending state, and a requirement to be fulfilled in order to enable the transition to take place, the identified transitions possibly including a transition that advances time,

step d) comprises:

– for each transition, generating a data structure representing the requirement to be fulfilled in order for the transition to be enabled,

step e) comprises:

– constructing a data structure representing the set of reachable states by:

* constructing a data structure R representing an initial state of the Petri net,
* repeatedly:
  - selecting a transition,
  - generating an amended data structure R' by conjugating the data structure representing the requirements of selected transition with R,
  - generating an amended data structure R'' by, in R', updating variables in accordance with the actions of the transition,
  - assigning R as the disjunction of R and R'',

  until R is unchanged for all transitions,

inquiring as to the existence of predetermined states of the Petri net using a method according to any of claims **37**, **38**, **39**, and **44**.

53. A method according to claim **12** for generating a data structure for analyzing a system modeled by a min/max/linear constraint model, the model having a number of nodes, each either being a "max" node, a "min" node or a "linear" node, and a number of constraints each pointing from one node to another, each constraint representing a time interval, comprising the steps of:

step a) comprises:

- determining a set variables, one for each node,

step b) comprises:

- identifying constraints between nodes, a constraint comprising a starting node, an ending node, and a time delay,

step d) comprises:

- for each node, generating a data structure by representing a relation between the actual node, the nodes from which constraints point to the actual node, time intervals of those constraints, and the type of the actual node (min, max, or linear),

step e) comprises:

- constructing a data structure by performing the conjunction of the data structures generated in step d).

54. A method of analyzing a system modeled by a min/max/linear constraint model, the method comprising:

- generating a data structure according to the method of claim **53** where the terminal nodes of the data structures are adapted to represent a "true" or a "false", and where the inequalities in the nodes are difference constraints,
- obtaining information from the data structure using the method of any of claims **37**, **38**, **39**, and **44**.

55. A method according to claim **12** for constructing a data structure for a system modeled as Boolean combinations of linear inequalities, wherein:

step d) comprises:

- determining the linear inequalities,
- defining a number of different expressions, each comprising a linear inequality, and

step e) comprises:

- combining the data structures using the method of claim **23**.

56. A method of analyzing a data structure constructed by the method of claim **55**, the method comprising altering the data structure using the method of any of claims **21**, **23**, **47** or **48** and performing an assessment according to any of the claims **37**, **38**, **39**, and **44**.

57. A method for analyzing an embedded system, a fault-tolerant system, a safety-critical system, or a concurrent composition of any such systems comprising

- modeling the system using a concurrent composition of timed automata,
- analyzing the model according to the method of claim **51**.

58. A method for analyzing an embedded system, a fault-tolerant system, a safety-critical system, or a concurrent composition of any such systems comprising

- modeling the system using a timed Petri net, and
- analyzing the model according to the method of claim **52**.

59. A method for analyzing an embedded system, a fault-tolerant system, a safety-critical system, or a concurrent composition of any such systems comprising

- modeling the system using a timed automaton, and
- analyzing the model according to the method of claim **50**.

60. A method for verifying interface timing between two components or systems, the method comprising:

- modeling the interface timing of the two components or systems using a min/max/linear constraint model,
- analyzing the model according to the method of claim **54**.

61. A method for analyzing economical systems, operations research systems, transport systems, or planning problems, the method comprising:

- modeling the system or problem using Boolean combinations of linear inequalities,
- analyzing the model according to the method of claim **56**.

62. A method for analyzing the timing behavior of a combinational circuit, the method comprising:

- modeling the gates of the circuit using a min/max/linear constraint model,
- analyzing the model according to the method of claim **54**.

63. A method for analyzing the timing behavior of a combinational circuit, the method comprising:

- modeling the gates of the circuit using timed automata,
- analyzing the model according to the method of claim **51**.

64. A method for analyzing the timing behavior of combinational parts of a sequential circuit, the method comprising:

- modeling the gates of the parts of the circuit using a min/max/linear constraint model,
- analyzing the model according to the method of claim **54**.

65. A method for analyzing the timing behavior of a sequential circuit, the method comprising:

- modeling the gates of the circuit using timed automata,
- analyzing the model according to the method of claim **51**.

66. A method for analyzing the timing behavior of an asynchronous circuit, the method comprising:

- modeling the gates of the circuit using a timed Petri net,
- analyzing the model according to the method of claim **52**.

67. A method for analyzing a sequential or concurrent computer program, the method comprising

- modeling statements, such as assignments or conditional guards, as expressions containing inequalities in a data structure as defined in any of claims **1–11**,
- achieving a model of the full program by:

- combining the models of the individual statements, using manipulation algorithms comprising Boolean operators, quantifiers and/or substitutions, according to any of the methods of claims **21**, **23**, **32–34**, **36**, **40–43**, and **45–49**,

- constructing a data structure R representing an initial state of the program,
- repeatedly:
  - selecting a statement,
  - generating an amended data structure R' by conjugating the data structure representing the requirements of selected statement with R,
  - generating an amended data structure R'' by, in R', updating variables in accordance with the actions of the statement,
  - assigning R as the disjunction of R and R'',

  until R is unchanged for all statements,
- analyzing the program by analyzing R using a method according to any of claims **37**, **38**, **39**, and **44** or performing an alteration according to any of claims **21**, **23**, **32–34**, **36**, **40–43**, and **45–49** and subsequently analyzing the program by analyzing the altered data structure using a method according to any of claims **37**, **38**, **39**, and **44**.

68. A data carrier comprising a data structure according to any of claims **1–11**.

69. A data carrier comprising a program for a computer, the program performing a method according to any of claims **12–67**.

70. A data carrier comprising a program for a computer, the program being adapted to enable a general purpose computer to perform the method according to any of claims **12–67**.

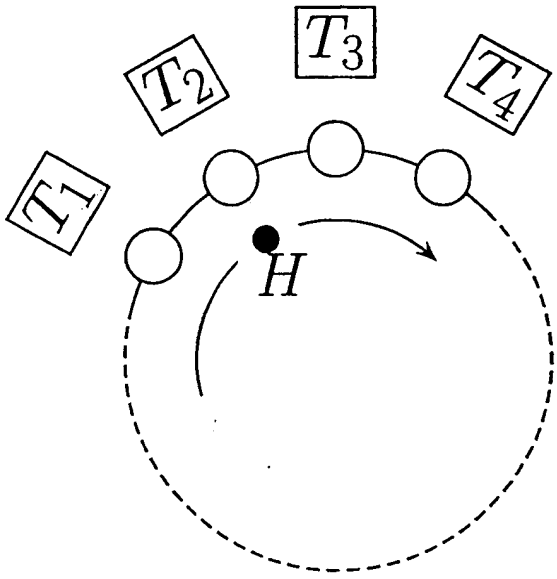


Figure 1.

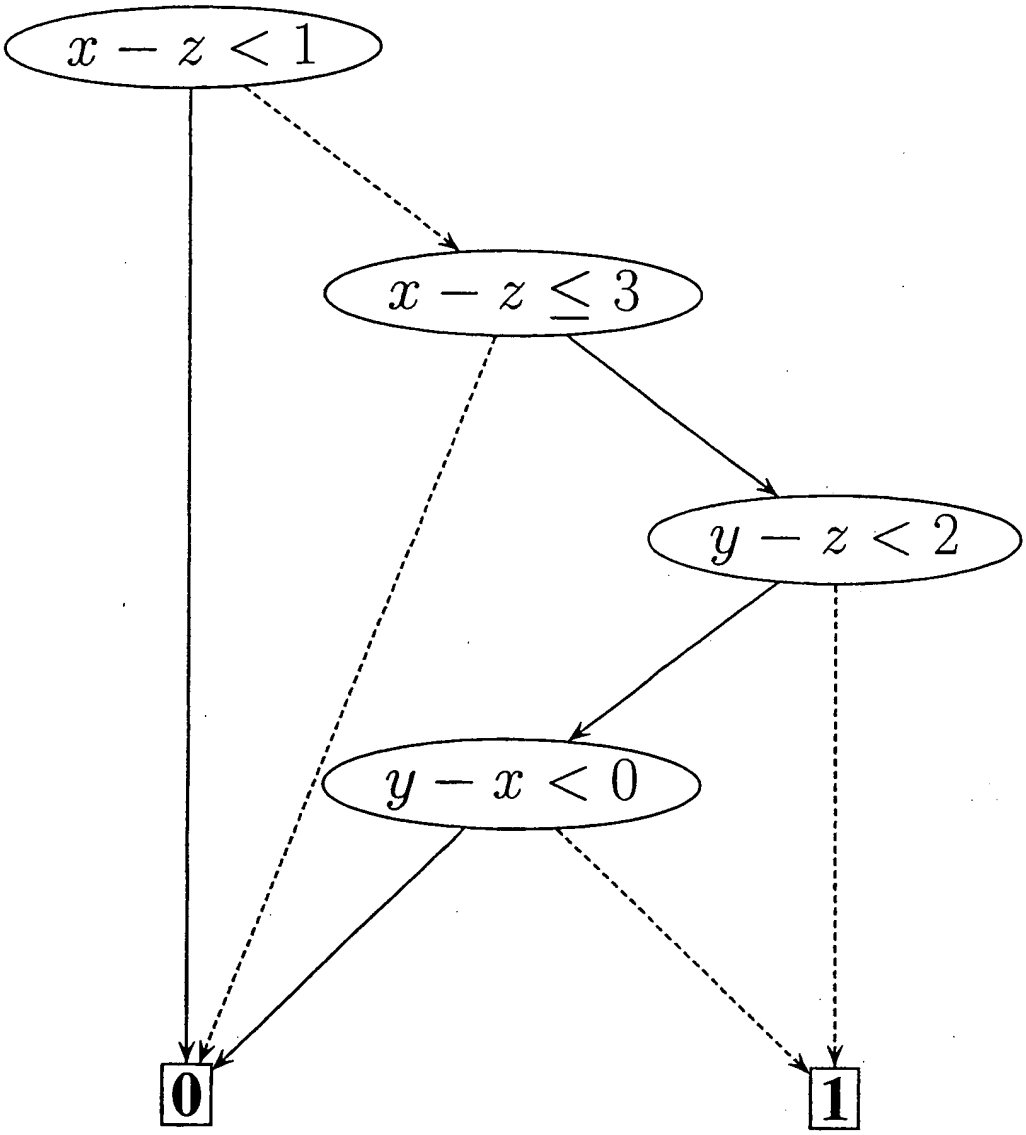


Figure 2.

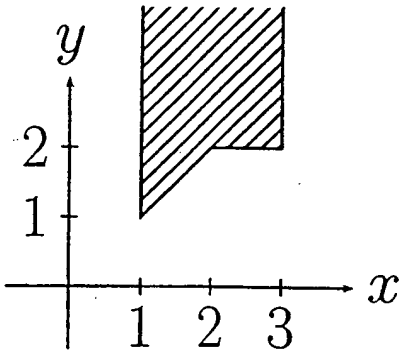


Figure 3.

N	KRONOS	UPPAAL	DDD
4	0.2	0.1	0.1
5	0.7	0.2	0.1
6	22.6	0.6	0.1
7	339.2	2.3	0.1
8	—	9.0	0.2
9	—	35.0	0.2
10	—	138.4	0.2
11	—	529.8	0.2
12	—	2560.7	0.3
16	—	—	0.5
32	—	—	2.2
64	—	—	15.9
128	—	—	123.3
256	—	—	1104.8

Figure 4.

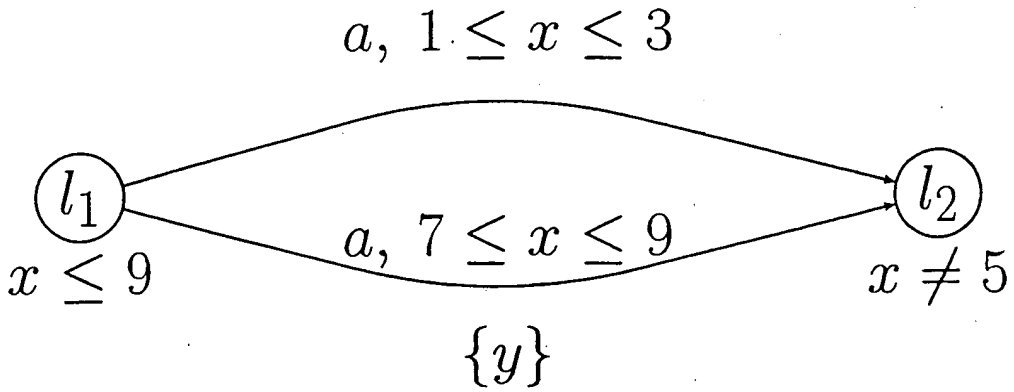


Figure 5.

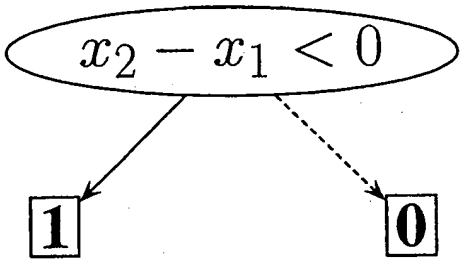


Figure 6.

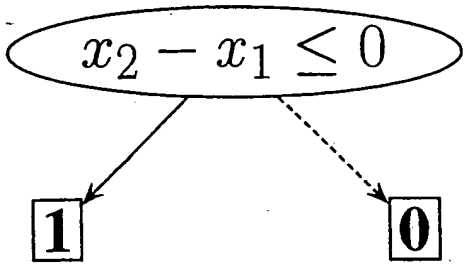


Figure 7.

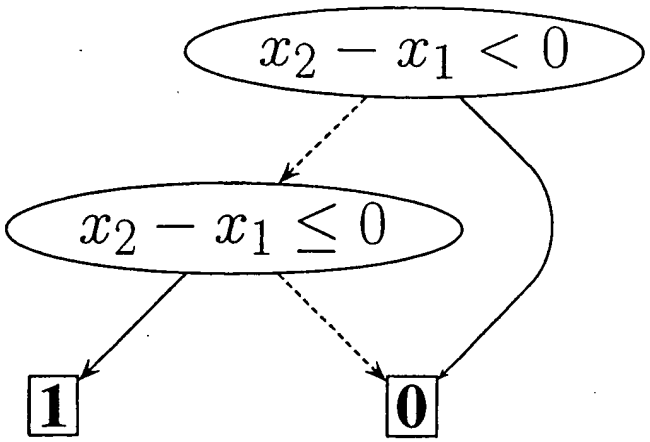


Figure 8.

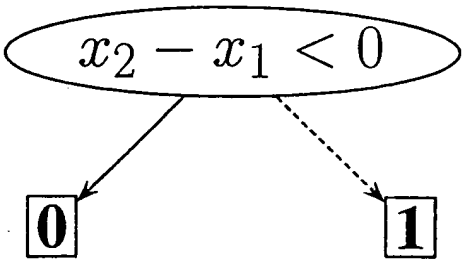


Figure 9.

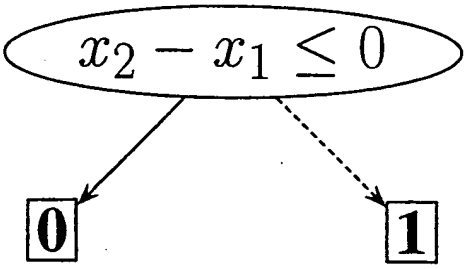


Figure 10.

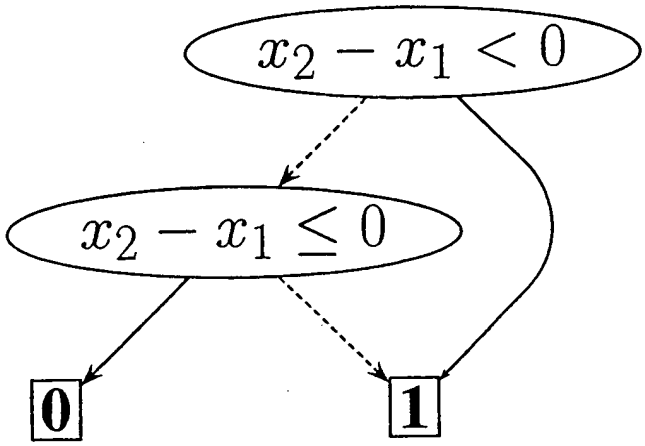


Figure 11.

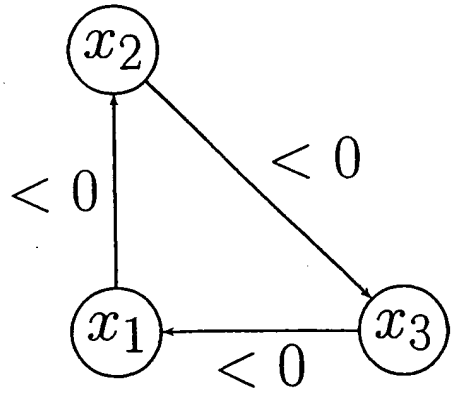


Figure 12.

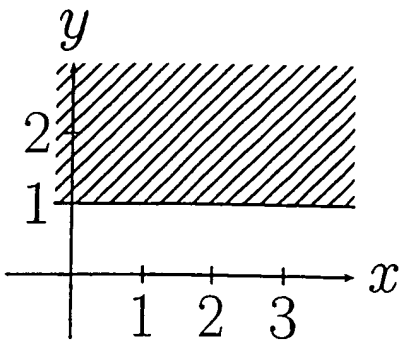


Figure 13.

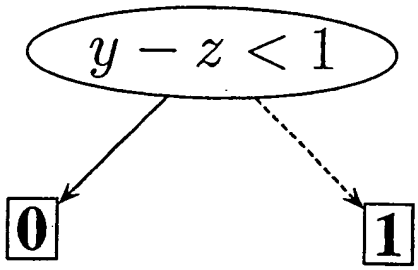


Figure 14.

INTERNATIONAL SEARCH REPORT

International Application No
PCT/DK 99/00456

A. CLASSIFICATION OF SUBJECT MATTER
IPC 7 G06F17/50 G06F17/60 G06F9/44

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
IPC 7 G06F

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practical, search terms used)

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category *	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X	JAIN P ET AL: "EFFICIENT SYMBOLIC SIMULATION-BASED VERIFICATION USING THE PARAMETRIC FORM OF BOOLEAN EXPRESSIONS" IEEE TRANSACTIONS ON COMPUTER AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS,US,IEEE INC. NEW YORK, vol. 13, no. 8, page 1005-1015 XP000467924 ISSN: 0278-0070	1-11, 37-39, 41-43,68
A	abstract	12,21, 35,36, 40,44, 45,47, 49,57-66
	page 1007, right-hand column, line 30 -page 1009, left-hand column, line 40	
	-/--	

[X] Further documents are listed in the continuation of box C.

[ ] Patent family members are listed in annex.

* Special categories of cited documents :

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier document but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art.

"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
30 November 1999	13/12/1999

Name and mailing address of the ISA	Authorized officer
European Patent Office, P.B. 5818 Patentlaan 2 NL - 2280 HV Rijswijk Tel. (+31-70) 340-2040, Tx. 31 651 epo nl, Fax: (+31-70) 340-3016	Kingma, Y

INTERNATIONAL SEARCH REPORT

Inte Jonal Application No
PCT/DK 99/00456

C.(Continuation) DOCUMENTS CONSIDERED TO BE RELEVANT

Category *	Citation of document, with indication,where appropriate, of the relevant passages	Relevant to claim No.
A	BURCH J R ET AL: "SYMBOLIC MODEL CHECKING FOR SEQUENTIAL CIRCUIT VERIFICATION" IEEE TRANSACTIONS ON COMPUTER AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS,US,IEEE INC. NEW YORK, vol. 13, no. 4, page 401-424 XP000453301 ISSN: 0278-0070 abstract page 403, left-hand column, line 13 -right-hand column, line 23; figure 1 ---	1
A	MILLER D M ET AL: "IMPLEMENTING A MULTIPLE-VALUED DECISION DIAGRAM PACKAGE" , THE INTERNATIONAL SYMPOSIUM ON MULTIPLE-VALUED LOGIC,US,LOS ALAMITOS, CA: IEEE COMPUTER SOC, PAGE(S) 52-57 XP000793445ISBN: 0-8186-8372-4 page 52, left-hand column, line 1 -page 53, left-hand column, line 38 ---	1,2
A	CHAN W ET AL: "MODEL CHECKING LARGE SOFTWARE SPECIFICATIONS" IEEE TRANSACTIONS ON SOFTWARE ENGINEERING,US,IEEE INC. NEW YORK, vol. 24, no. 7, page 498-519 XP000781149 ISSN: 0098-5589 abstract page 509, left-hand column, line 17 -page 510, left-hand column, line 53 ---	1,67
A	JIN YANG ET AL: "Symbolic model checking for event-driven real-time systems" ACM TRANSACTIONS ON PROGRAMMING LANGUAGES AND SYSTEMS, MARCH 1997, ACM, USA, vol. 19, no. 2, pages 386-412, XP002124332 ISSN: 0164-0925 abstract page 408, line 10 -page 411, line 19 -----	1,57,59, 60,62-65